(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
16 May 2002 (16.05.2002)

PCT

(10) International Publication Number
WO 02/39325 A2

(51) International Patent Classification[7]: G06F 17/50

(21) International Application Number: PCT/BE01/00193

(22) International Filing Date:
8 November 2001 (08.11.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
0027258.3 8 November 2000 (08.11.2000) GB

(71) Applicant *(for all designated States except US)*: **EASICS** NV [BE/BE]; Interleuvenlaan 86, B-3000 Leuven (BE).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: **VANDEWEERD, Ivo** [BE/BE]; Vuurkruisenlaan 1, B-3500 Hasselt (BE). **COENEN, Steven** [BE/BE]; Grootloonstraat 101, B-3840 Borgloon (BE).

(74) Agents: **BIRD, William, E.** et al.; Bird Goën & Co., Vilvoordsebaan 92, B-3020 Winksele (BE).

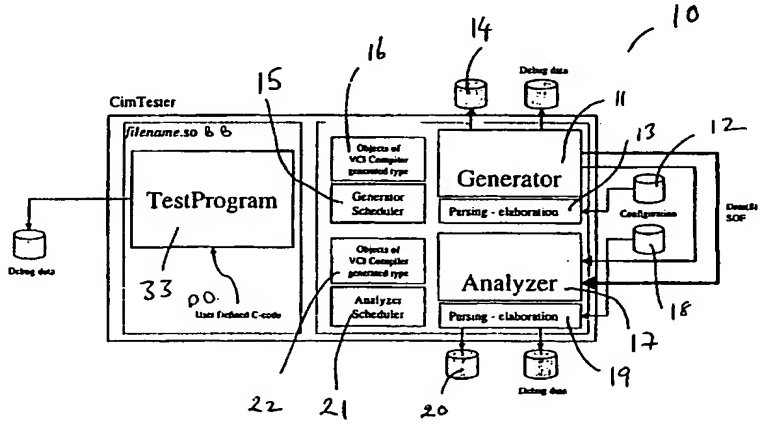
(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Declaration under Rule 4.17:
— *of inventorship (Rule 4.17(iv)) for US only*

[Continued on next page]

(54) Title: COMPUTER BASED VERIFICATION SYSTEM FOR TELECOMMUNICATION DEVICES AND METHOD OF OPERATING THE SAME



(57) Abstract: A computer apparatus is described for displaying and manipulating sequences of frames of hierarchically organized framed data. The apparatus comprises means for generating a graphical user interface on a display screen, the graphical user interface having means for displaying a representation of a hierarchy of the framed data. The computer apparatus also has a pointing device for selecting a portion of the representation of the hierarchy to generate a control signal, and means responsive to the control signal to display a portion of the framed data corresponding to the selected portion of the representation of the hierarchy. The graphical user interface may be used in a computer based verification system for the analysis and display of a sequence of frames of framed data, comprising: means for generating a frame generator for receiving the frame sequence, the frame generator comprising means for processing the frame sequence in accordance with a protocol to form a modified frame sequence and for associating with each frame a description data structure for that frame, and means for outputting the modified frame sequence and the frame description data structure.

WO 02/39325 A2

Published:

— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

COMPUTER BASED VERIFICATION SYSTEM FOR TELECOMMUNICATION DEVICES AND METHOD OF OPERATING THE SAME

The present invention relates to a verification system, in particularly a test bench system and method for verifying telecommunication system component and component design conformance with protocols and well as a usr graphics interface to be able to view traffic data manipulated by the verification/test bench system or method.

TECHNICAL BACKGROUND

Computer and telecommunications technology has created a huge potential for integration. The design community has tackled a variety of subjects that try to make the maximum use of this potential by mastering this increased complexity. A crucial component in these efforts, is the challenge to overcome the *verification complexity.* Verification of telecommunications and computer based systems and components to be used in these systems is still a most time consuming and critical aspect of system design. Generally, all such systems and components operate in accordance with a certain protocol. A protocol is a set of rules which relates events of the past and/or present to those of the future, i.e. it defines action/reaction or stimulus/response behavior of the system. Complex telecommunications systems, such as a mobile telephone network or a Wide Area Network such as the Internet involve many individual components and/or complex protocols. All these network elements must conform to every aspect of the specified protocols. Generally, there are two major problems: a) to confirm that a specific design of a component conforms to the protocols, and b) a specific piece of equipment conforms to the protocols. Problem a) is often solved by simulation and b) by means of test benches. However, both solutions share a problem that any comprehensive test must include not only challenges with correct data but also ones with erroneous data, i.e. response behavior to both correct and erroneous stimuli must be checked. When an error is found, the designer is faced with a laborious process of finding out where the problem lies. This generally involves sifting through the recorded behavior of the object under test. The problems with test benches may be summarized as:

–Testbench creation is at least as complicated as the actual design of a component,

most of the time more complicated. A testbench has to be able to create all possible kinds of correct traffic but also a whole variety of incorrect traffic. The incorrect traffic is used to see if the design reacts and reports this incorrect traffic correctly.

– Testbenches are duplicated over and over again for different projects. Most testbenches are too project specific so that they cannot easily be reused in a new project.

– Testbenches that are written by the same team that designs a component such as a chip, lack a cross-check with an external source. A misinterpretation of the specification may go unnoticed.

One object of the present invention is to provide a verification or test bench environment and method of operating the same which allows rapid testing of designs and equipment.

Another object of the present invention to provide a verification or test bench environment and method of operating the same which allows easy visualization of the results.

SUMMARY OF THE INVENTION

In one aspect, the present invention provides a design verification environment, e.g. a test bench environment, which combines a multi-project platform approach with a modular, extendible and high-level description capability required for telecommunication designs and devices such as complete systems-on-chip.

The present invention provides a computer apparatus for displaying and manipulating sequences of frames of hierarchically organized framed data, comprising:

means for generating a graphical user interface on a display screen, the graphical user interface having means for displaying a representation of a hierarchy of the framed data,

a pointing device for selecting a portion of the representation of the hierarchy to generate a control signal, and

means responsive to the control signal to display a portion of the framed data corresponding to the selected portion of the representation of the hierarchy.

The present invention may also provide a computer based verification system for the analysis and display of a sequence of frames of framed data, comprising:

means for generating a frame generator for receiving the frame sequence, the frame generator comprising means for processing the frame sequence in accordance with a protocol to form a modified frame sequence and for associating with each frame a description data structure for that frame, and

means for outputting the modified frame sequence and the frame description data structure.

The present invention may also provide a method of analyzing and displaying a sequence of frames of framed data, comprising the steps of:

generating a frame sequence in accordance with a protocol,

associating with each frame a description data for that frame, and

outputting the frame and the frame description data.

The present invention also comprises a method of analyzing framed data comprising the steps of: entering a descriptor file at a near location such as a terminal connected to a telecommunications network, the descriptor file comprising at least a representation of the operation of a telecommunications device;

transmitting the descriptor file over the telecommunications network to a remote location;

generating framed data at the remote location using a computer based system in accordance with any of the methods of the present invention;

and receiving at a near location the modified framed data and description data associated with the frame data.

The present invention also includes computer program products for executing any of the methods of the present invention when executed on a computer.

The present invention will now be described with reference to the following drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Figs. 1a and 1b are schematic representations of a verification system in accordance with embodiments of the present invention. Fig. 1a shows a standalone embodiment and Fig. 1b shows a co-simulation embodiment.

Fig. 2 is a schematic representation of a generator for use with the present invention.

Fig. 3 is a schematic representation of an analyzer for use with the present

invention.

Fig. 4 shows the relationship between a common interface, a generator and an analyzer interface.

Fig. 5 is a schematic representation of a block communication means for use with the present invention.

Fig. 6 is a schematic representation of how blocks can be connected by a block communication means in the present invention.

Figs. 7a and 7b show how more than one block may be connected to a generator and an analyzer respectively.

Figs. 8a and b show linked generators and linked analyzers, respectively, in accordance with an embodiment of the present invention.

Fig. 9 shows a generator tree as used with the present invention.

Fig. 10 shows a generator scheduling as used with the present invention.

Fig. 11 shows an analyzer tree as used with the present invention.

Fig. 12 shows an analyzer scheduling as used with the present invention.

Fig. 13 is an overview of a standalone verification scheme in accordance with an embodiment of the present invention.

Fig. 14 is an overview of a co-simulation verification scheme in accordance with an embodiment of the present invention.

Fig. 15 shows a scheduling cycle for a standalone verification system in accordance with an embodiment of the present invention.

Fig. 16 shows a scheduling cycle for a co-simulation verification system in accordance with an embodiment of the present invention.

Fig. 17 shows the generation of descriptor data for frames during data stream generation in accordance with an embodiment of the present invention.

Fig. 18 shows the generation of descriptor data for frames during data stream analysis in accordance with an embodiment of the present invention.

Fig. 19 shows a screen shot of the display of a user graphics interface in accordance with an embodiment of the present invention.

Figs. 20a shows a screen shot of the data displayed in Fig. 19 for a different layer of the hierarchy.

Fig. 20b shows a detailed screen shot of the display of a user graphics interface in accordance with an embodiment of the present invention.

Fig. 21 shows a schematic representation of a computer with which the present invention may be used.

Fig. 22 shows a tagging scheme in accordance with the present invention.

ACRONYMS

ATM Asynchronous Transfer Mode
DUT Design Under Test
HDL Hardware Description Language
PHY Physical layer
UTOPIA Universal Test & Operations
PHY Interface for ATM
VCI Virtual Component Interface
VHDL VHSIC Hardware Description Language
VHSIC Very High Speed Integrated Circuit

DETAILED DESCRIPTION OF THE PRESENT INVENTION

The present invention will be described with reference to certain embodiments and drawings but the present invention is not limited thereto but only by the attached claims. For example, the present invention will mainly be described with reference to the computer language C++ but other languages may be used, e.g. ADA.

The present invention provides a computer-based design verification system, in particular a test bench system for generating and analyzing telecommunication data traffic, especially as related to a design of a telecommunications device or a component thereof. The verification or test bench system may be implemented in hardware but in a preferred embodiment the verification or test bench system is implemented in software, e.g. computer program products written in a suitable computer program language such as C++ and their execution on a computing device. The computer program product when run on a suitable computing device generates and analyzes telecommunication data traffic. The verification or test bench system may find advantageous use as a reference system or test bench to verify the functional correctness of designs or components of telecommunications systems such as complete devices or designs of components used in such devices.

The verification or test bench system comprises a frame generator and an analyzer, preferably implemented as a multi-protocol test bench for traffic data simulation at the system level. It is not anticipated that the nature of the telecommunications protocol will be a limitation on the present invention, e.g. suitable protocols are PDH, SDH, SONET, ATM, HDLC, PPP, IP. The verification or test bench system also includes a graphical user interface for viewing and manipulating the data which has been generated and/or analyzed. Optionally, the present invention also includes verification and analysis of physical devices using the verification system.

The verification system can be operated as a standalone system 10 and is shown schematically in Fig. 1a. The system 10 is for use on a general purpose computer and the verification system is implemented as a computer program running on the computer, e.g. as shown in Fig.21. The main elements of the system are:

1. One or more data generators 11 for generating a stream of digital data in accordance with one or more protocols. The structure and function of these generators is determined by a configuration file 12 which is a descriptor file which may be read by the computer program running on the computer, parsed by a parsing element 13 and the generators configured along with their communication links on instantiation of the system. For instance, the configuration file may be in XML format. The data stream generated is associated with descriptor data for the stream which will be described later. This descriptor data can be dumped and stored in memory 14, e.g. on a hard disk, a CD-ROM, diskettes, tape, etc. To control the functioning of the generators 11, a generator scheduler 15 is provided. Optionally, an object generator 16 may be provided for generating objects such as ones of the VCI compiler type.
2. One or more data analyzers 17 for analyzing a stream of digital data in accordance with one or more protocols. The structure and function of these analyzers is determined by a configuration file 18 which is a descriptor file which may be read by the computer program running on the computer, parsed by a parsing element 19 and the analyzers configured on instantiation of the system along with their communication links. For instance, the configuration file may be in XML format. The data stream analyzed is associated with descriptor data for the stream which will be

described later. This descriptor data can be dumped and stored in memory 20, e.g. on a hard disk, a CD-ROM, diskettes, tape, etc. To control the functioning of the analyzers 16, a generator scheduler 21 is provided. Optionally, an object generator 22 may be provided for generating objects such as ones of the VCI compiler type.

3. A test program 23 which can be generated by a user and which determines the tests to be carried out. The test program is read by the verification system and executed.
4. Optionally, to provide communications with a design to be tested, address locations for registers in the design to be tested may be generated automatically, e.g. by VCI compilers 16 and 22 for the generators and analyzers respectively.

A verification system can also be operated as a co-simulation system 10 and is shown schematically in Fig. 1b. Items with the same reference numbers as in Fig. 1a refer to the same items. The main difference is that a simulation 26 of the design to be verified is also run on the computer and communicates with generators and analyzers via a suitable interface, e.g. a PLI interface, 27, implemented as a computer program or routine running on the computer.

Further, actual physical devices can be verified by using the verification system 10 of Fig. 1a. Firstly, the standalone system 10 is run using a simulation of the device to be checked to generate a data stream as well as data descriptor files stored in memories 14 and 20. The generated data stream is then supplied to the device to be tested. The output of the device is recorded on a suitable medium or delivered directly to the verification system 10. This device output is input into the system 10, namely to the analyzers. These analyze the data based the data file in memory 20 from which a frame organization can be derived. Errors can then be determined and examined, e.g. by displaying the traffic data frames using a user graphics interface preferably in accordance with the present invention (described later).

Provided the computer system can deal with the realtime restraints of a physical device, the output of a real device may be connected directly to the computer, e.g. via a suitable interface and if required via a buffer in case rate matching between the computer and the device are required. In this case the data stream output of the generators 11 can be fed to the device through a suitable interface and its response

delivered to the analyzer section of the verification system again through an interface. The frame data descriptor file generated by the generators is provided to the analyzers so that they demultiplex the received data and allow visualization thereof.

Typically, telecommunications systems use data in frames, whereby the frames are organized into a hierarchy, e.g. of frames, super frames, etc. In embodiments of the present invention a collection of traffic data generators and analyzers are provided that can be linked together to create or analyze a specific data stream, respectively. A block may be either a generator or an analyzer. Each type of block implements a communications protocol or a part of such a protocol. By connecting different instances of various types of blocks, a more complicated hierarchy of protocols, or one big protocol with different levels can be created. How to connect all the blocks is specified in a configuration file as described above. Both generator side and analyzer side each have a configuration file. Multiple, possibly different, trees of hierarchy can be created in one configuration file. This can be useful if the hardware design supports multiple protocols.

A computer based verification or test bench system in accordance with the present invention stores a parseable description of how different protocol layers can be structured and preferably includes validation means to check if a configuration file is in violation with one of the structuring rules and to throw an error message if this is the case. Preferably, the verification or test bench system includes means to provide a suggestion on how to make the structuring valid.

Each block in a verification or test bench system according to the present invention has control variables that can alter the behavior of a block or even change the protocol generated or analyzed by that block. Inserting protocol errors is an example of a change in behavior. Each block also has means for giving information about the current state of the block. An error counter is a typical example of such a means. To set and read all these control variables and status registers, a user can create a test plan function written in a suitable computer program language such as C++. This test plan function has to be compiled into a shared object file. The test bench system loads this shared object file and executes the compiled test plan function when the simulation starts. In the test plan function, the user can use wait statements to pause the execution of the test plan function for a number of cycles or until a certain event occurs. A verification or test bench system according to the present invention puts no

restrictions on the test plan function, i.e. the user can write any code desired. User customization can include creation of a customized graphical interface, reading in and parsing a command file for stimulating some control signals, integration of third party libraries.

Not only the blocks of a verification or test bench system according to the present invention have control and status registers, a device design under test (DUT) with the test bench according to the present invention may also have several control and status registers. The present invention includes a method for hardware/software cosimulation, i.e. the ability to access control and status registers from a computer program. This computer program can generate a test bench to test the DUT on execution on a computing device or can contain the code that will run on an embedded processor. This way of accessing registers in the DUT can also be used. The present invention includes reactive test plans. For example, the user can insert an error in a generator using a control signal from that generator and then check that the DUT has processed that error correctly by monitoring the status registers of the DUT.

Sometimes a device design to be tested has its own proprietary protocol for internal use. A test bench system in accordance with the present invention may optionally be extended with user defined protocols in the form of an external block. Such an external block is treated the same way as a built-in block. All the default capabilities of a built-in block are made available to an external block.

A verification or test bench system in accordance with an embodiment of the present invention may comprise three main parts:

– Part 1 sets up the complete running environment. It preferably includes means for reading configuration files, means for validating them, means for creating all blocks and means for interconnecting all the blocks. In design cosimulation with a simulator such as an HDL simulator, this part also includes means for setting up a connection to the simulator.

– Part 2 is a scheduler. This part makes sure the correct block is executed at the correct time.

– Part 3 comprises the blocks, the generators and the analyzers of the verification or test bench system. These items implement protocols.

The running environment setup part when executed on a computing device executes the same sequence of tasks every time the test bench system is started:

1. Parse command line arguments.

2. Parse a settings file if it exists

3. Read generator and analyzer configuration files

4. Validate the configuration file, i.e. check that the structuring of blocks in the configuration file is valid.

5. Create and connect generator blocks with their configurations

6. Create and connect analyzer blocks with their configurations

If the test bench system is running in cosimulation with a simulator such as an HDL simulator, this part also sets up the connection with the simulator. It also instructs the scheduler to use the connection with the simulator, that is to send generated data to, and read data to be analyzed from the simulator.

All the blocks in the verification or test bench system preferably have a common interface 30. For generators and analyzers, this common interface 30 has generator or analyzer specific properties, respectively.

The common interface 30 for all the blocks consists of properties that all blocks share in the verification or test bench system. The verification or test bench system operates with and on framed digital data. Among these common properties the following is a non-limiting list: byte, row, column and frame counters, size and shape of a frame in a protocol, an indication if the protocol is bit oriented or byte oriented, a method to reset a block. The use and implementation of these properties is protocol specific and therefore implemented differently for each protocol. Some properties are the same for many protocols. For those very common properties, a default implementation can be provided. For example, the shape of a frame of data in a protocol is often rectangular. The common interface can therefore provide functions to calculate the byte, row, column and frame counters for rectangular frames.

The interface 31 for a generator preferably comprises:

– A list of input traffic streams to the generator. The size of the list can be 0 which means that the generator does not have an incoming data stream. The size of the list is not fixed, it depends on the protocol. For example, for a protocol that multiplexes 3 data streams into 1 data stream the size of the list would be 3.

– One output of the generator. All the traffic that a generator generates will be presented on this output.

– A function that runs the implementation of the protocol, i.e. generate traffic and put

it on the output.

All generators share this generator interface 31. Each generator can add other protocol specific properties when needed. A schematic representation of a generator 11 is shown in Fig. 2.

The interface 32 for an analyzer preferably comprises:

– One input of the analyzer. All the traffic that an analyzer analyzes will be available on this input.

– A list of output traffic streams. The size of the list can be 0 which means the analyzer does not have an outgoing data stream.

– A functions that runs the implementation of the protocol, i.e. analyze the traffic available on the input.

All analyzers share this analyzer interface 32. Each analyzer can add other protocol specific properties when needed. A schematic representation of an analyzer 17 is shown in Fig. 3.

The relationship between a common interface 30 and specific generator and analyzer interfaces 31, 32 can be seen in Fig. 4.

The inputs and outputs for both generator and analyzers also have the same interface and the same implementation. In a verification or test bench system in accordance with the present invention these inputs and outputs are called block communication means 34. Block communication means 34 provide means to put data onto or get data from a block. Also means are provided to check if there is still room to store data on a block or to see if there is data available. Besides that, the block communication means 34 also provide means and methods to annotate traffic data with attributes. On example of an attribute of data is e.g. whether or not this particular data is the start of a frame. A graphical view of the block communication means 34 is shown in Fig. 5.

To connect two blocks, one block communication means 34 has to be created. The output from the first block and the input of the second block have to be connected to the block communication means 34. Because all the inputs and outputs of all the blocks have the same standardized interface – i.e. block communication means 34 - any block can be connected to any other block. Fig. 6 shows schematically how this is done.

Generators can have multiple inputs. The input side of the generator can be

connected to multiple other blocks. An example is shown in Fig. 7a.

Analyzers can have multiple outputs. The output side of an analyzer can be connected to multiple other blocks. An example is shown in Fig. 7b.

Mixing generators and analyzers can be provided in accordance with the present invention. Preferably, in a test bench system in accordance with the present invention traffic data is generated for a DUT and traffic from the DUT is analyzed.

A complete scheme for generators and for analyzers is shown in Fig. 8a and 8b respectively. Fig. 8a shows a hierarchy of generator blocks which finally produce a stream of framed data from the final generator (a scheme for the SDH is shown by way of example). Each generator implements a part of the total protocol. The aim is to generate a data stream which conforms to the standard or which has been modified in a specific way so as to challenge a design under test. Effectively, the data generated by the generators is multiplexed together to form a single data stream. To provide some realistic data various data sources can be provided. Framed data usually consists of user data or "payload", and administrative data called "overhead". A suitable payload can be read (FileRead) for instance from a file in memory in which sample payload data is stored. Random data may also be generated in a random generator (PRBS). Fig. 8b shows a similar hierarchical structure for analyzers. In this case there is a single bit stream input and this is segmented to obtain the individual overhead and payload information, effectively by demultiplexing. Further data sinks are provided, e.g. the data generated may be stored in a file (FileWrite), data which should be random can be checked by a random number checker (PRBS).

The scheduling for generators and analyzers is different. As shown above an interconnection of multiple generators is generally organized in a tree structure. There is 1 top level generator with 0, 1 or more generators connected to it. Each connected generator can have 0, 1 or more generators connected to it, just like the top level. The resulting structure a tree as shown schematically and in more detail in Fig. 9. The generation of traffic data is started whenever traffic is requested from a block communication means 1. Block communication means 1 knows from which block it should request traffic when it has to generate traffic itself. In this case this is the top level generator. It requests traffic from that generator by calling the function that generates traffic and puts that traffic on its output. The output of the generator is the same block communication means as the one that requested the traffic. So the block

communication means requests traffic from the generator and the generator delivers that traffic to the block communication means. The generator itself may be able to generate the requested traffic by itself or may need traffic from another generator. Which option the generator has to take depends on the protocol. If the generator can generate the traffic by itself, the end of the scheduling cycle has been reached. The request for traffic on block communication means 1 has then been fulfilled. If the generator needs traffic from another, lower level, generator, the same algorithm is used. The generator requests traffic from block communication means 2 or 3, depending on the protocol. Suppose its from block communication means 3. Then block communication means request traffic from the generator that is connected to it (the left one) in the same way as the top level block communication means requested traffic from the top level generator. This scheduling can ripple to all the leaves of the tree, or it can stop somewhere in the middle. The leaves of the tree are special generators G*, they don't need traffic from another generator, they can generate all traffic by themselves, i.e. they are data sources. They may be referred to as Data Generators. Fig. 10 shows a scheduling for generators.

Note that, in accordance with embodiments of the present invention, generators don't have to generate traffic when they are not doing anything. This differs, for instance, from an HDL simulator where each logical block runs on a clock. On every event of that clock, a logical block is executed, even if there is no need to execute it. The HDL simulator has no way to schedule the block intelligently because it has no information on the block relating to how to schedule it. In accordance with the present invention the scheduler is configured as an "on-demand" scheduler, that is the scheduler does not activate blocks at each time step but only activates blocks when these have a function to carry out.

The block communication means 34 preferably also has some built-in intelligence. If a generator did not generate enough traffic for the request, a new request is sent to the generator to generate more traffic. This is done until the generator has generated enough traffic to fulfill the request. For example, if a generator implements a bit oriented protocol, the generator may generate 1 bit at a time. If the generator is connected to another generator that implements a byte (= 8 bits) oriented protocol, meaning it always sends out request for bytes, the block communication means will make sure that a complete byte is always generated. It does this by

requesting 1 bit from the bit oriented generator, 8 times in a row.

An interconnection of multiple analyzers is also in a tree structure. There is 1 top level analyzer with 0, 1 or more analyzers connected to it. Each connected analyzer can have 0, 1 or more analyzers connected to it, just like the top level. Fig. 11 shows an example.

The scheduling algorithm is similar to that for a generator, but the request is different. For analyzers, the request is a request to analyze traffic. A block communication means has a certain traffic data and sends a request to the analyzer it is connected to, to analyze that traffic data. The analyzer retrieves the traffic data from the block communication means and analyzes it. The analyzer can either analyze the traffic itself, or it can pass the traffic to another analyzer requesting it to analyze the traffic. Fig. 12 shows how this is done.

Most of the time, the standard behavior of a block is not sufficient to test a DUT. The present invention allows a user to modify the behavior of a block to get a specific behavior of a block, i.e. the present invention includes means to modify the behavior of a block. The block may also have means to report some information to the user. All this information and configuration is grouped together in a structure. For each type of block (implementing a specific protocol) this structure is different. For two instances of the same type of block, there are also two instances for this structure. Both the user and the block can easily access these structures. These structures are important for the user. They give information about what a block in the verification or test bench system is capable of doing. Because of the importance of the structures, they are preferably well documented and easy to understand. In the verification or test bench system the structures for each type of block may be described in a 'datamodel' file. From that file, computer code is extracted which implements the structures. This way, developers can very easily write documentation for the block they are developing. The document that is extracted is called the 'Control signals' document.

Because the structures are generated separately, they are preferably kept separate from the blocks. When setting up the running environment, the test bench system creates the structures when needed and attaches them to the correct blocks. A reference to the structure is also stored for use in a test program.

The test program is the part the user has to write or standard test programs may be generated. It controls all the blocks in the test bench system. In the test program, the

user can also control the DUT through an interface such as a VCI interface. The user can also write wait statements in the test program. This suspends the test program for a certain time, while the generators and analyzers keep on running. How long the test program is suspended depends on the type of wait statement used. There can be 3 types of wait statements, for instance:

- WAIT_CYCLE suspends the test program for 1 cycle.
- WAIT_CYCLES(n) suspends the test program for n cycles.
- WAIT_UNTIL(condition) suspends the test program until the given condition becomes true.

The generator tree and the analyzer tree can become very large. It's also possible and very likely that there are many generator instances in the tree of the same type (implementing the same protocol). The present invention provides an easy way to uniquely identify each generator in the tree. A simple tagging scheme used for this identification :

- The top level block in the tree gets tag '0'.
- All the children of a block inherit the tag of their parent. For each child, '.n' is added to the parent tag where n is the number of the child. Fig. 22 shows this tagging scheme. If the user wants to change some configuration of a block, the tag of the block has to be known. The verification or test bench system can provide a list of all the tags on request. Once the tag is known, the user can change the configuration with a line of code like :

VC_DataGenerator["tag"].config.mode = CT_INCREMENTAL;

VC_DataGenerator is the name of the configuration for the data generator block. All the names of the configurations are documented in the 'Control signals' document. "tag" is the string representation of the tag associated with the block.

Some combinations of protocols require more communication than just passing traffic data. Sometimes the need arises to negotiate on what kind of traffic should be sent. To be able to implement such protocols in a verification or test bench system according to the present invention, a message protocol is provided to communicate between blocks. A block has means to send a message to another block that is attached to it. The block that receives this message, has to process it and send back a reply. If a message is not processed, the test bench system will give an error message.

When executed on a computing device the test program preferably has one

thread of control. If the user has various events that occur at different times, managing these events in a single thread of control requires care.

In a verification or test bench system according to the present invention, the user can define a computer program function that has to be executed every cycle, the CycleFunction. It also has to finish in the same cycle, so wait statements are not allowed in this function. In this function, the user can check for events and report/store information for the main test program. Several events can be checked and reported/stored in this function.

Another option to check various events is to have multiple test programs running in parallel in a multi-threading environment.

This CycleFunction concept is further refined in CycleObjects. The user can attach an object to a block. Before the block will generate or analyze traffic data (after a request to do so), a specific function in this cycleobject is called. That function can check various things in the block or somewhere else and report/store information for the main test program or for some other CycleObject. Because this is an object (and not just a function), it can store some state inside itself. What is stored is totally up to the user. This can be useful when the user is e.g. counting events. The counter can be stored in the object. As with the CycleFunction, the function in the CycleObject cannot have wait statements.

The configuration and status of a block only contains variables that can be set and read. Correctly setting a complicated configuration can easily become very cumbersome. A higher level of configuring a block can be useful in this case. A higher level means not setting a plurality of variables in a consistent way, but calling a function that does the setting of the variables and keeps them consistent. The best place to put such a function in inside the block the user is configuring because that is also the block that will be using the settings. To be able to do this, the user has to be able to access all the blocks in the test bench system. Once access to the block is achieved, the user can call the functions as required. Accessing a block can use the same tagging scheme as for accessing the configuration structures. e.g. :

Block_ATMTrafficSplitterAnalyzer["tag"].addRoute(3,5,1);

Block_ATMTrafficSplitterAnalyzer gives access to the correct block based on the tag.

addRoute is a function defined and documented in the ATMTrafficSplitterAanlyzer.

A verification or test bench system in accordance with the present invention can have two modes of operation.

– a standalone mode, i.e. without a simulator attached to it. The traffic generated by the generator tree is fed into the analyzer tree.

– cosimulation mode, i.e. with a simulator attached to it. The traffic generated by the generator trees is transferred to the simulator domain and is there applied to the DUT. Traffic coming from the DUT that is to be analyzed by the test bench system is transferred to the test bench system domain and fed into the appropriate analyzer tree.

The overall architecture in standalone mode is shown if schematically in Fig. 13. In this system the output of the generators is fed to the input of the analyzers. In step 1 the configuration files are read and parsed, in step 2 the test program is loaded including CycleFunction and CycleObjects, in step 3 the generators and analyzers are created as well as all the block communication means. At this point the verification system has been created. Traffic generation is initiated in step 4 by the main loop scheduler requesting traffic from the generator tree and requesting the analyzer tree to analyze that traffic. This is one cycle. This process is continued until a given cycle limit is reached.

The overall architecture in cosimulation mode is shown schematically in Fig. 14. A DUT can support multiple protocols, so in cosimulation the present invention supports multiple generator and analyzer trees. Operation is similar to the standalone mode but the output from the generators is sent to the DUT via the simulator interface and received therefrom for the analyzers. Alos an 'Extra Command Interface' may be used when the test bench system domain needs some information from the simulator domain that is not directly related to the traffic being generated or analyzed. This interface is also used to do some settings in the test bench system cosimulation library. For example, the user can request the current simulation time in the test program without advancing time in the simulator using a get function.

The communication between the simulator and the test bench system program is preferably implemented using 2 unidirectional pipes per type of communication. There are 4 types of communication (generator traffic, analyzer traffic, VCI communication and 'Extra Command Interface' communication), so 8 unidirectional pipes are used. Both the test bench system and the cosimulation library monitor all pipes to see if there are requests coming on a pipe. If there is a request on a pipe, it is

processed and all pipes are being monitored again.

Note that the test bench system and the simulator are in lock-step, i.e. when test bench system is running the simulator is paused. When the simulator is running, the test bench system is paused. This gives fully deterministic execution. Running the same simulation twice gives exactly the same results.

In cosimulation mode, it is the simulator that drives traffic generation and analysis. Each generator tree and analyzer tree is associated with a clock in the simulator. On every rising edge of such a clock, a request is sent to the appropriate generator or analyzer. For a generator, the test bench system will generate traffic and the cosimulation library will put that on the signals connected to the DUT. For an analyzer, the cosimulation library reads the traffic available on the signals coming from the DUT and transfers that traffic to the correct analyzer.

The test bench system presents/expects the traffic to follow a protocol. If the DUT does not have the exact same protocol, a small module that converts the protocols can be used. This converter module is called an adaptor.

A scheduling cycle is as follows:

1. A test program is executed until a WAIT_CYCLE is encountered. The other wait statements (WAIT_CYCLES(n) and WAIT_UNTIL(condition)) both use WAIT_CYCLE.

2. The WAIT_CYCLE activates the main loop scheduler.

3. The main loop scheduler is different in standalone mode and cosimulation mode

In the Standalone mode the CycleFunction is executed. The generator tree get's a request to generate traffic (1 byte of data). That traffic is given to the analyzer tree to analyze. The current cycle count is compared with the given cycle limit. If it is greater, the verification system terminates. If it is not greater, the main loop scheduler finishes and the testplan execution is resumed.

In cosimulation mode the scheduler watches the communication pipes with the simulator. There are 3 possible cases:

– If a request comes for generator traffic (rising edge of a generator clock), the correct generator tree get's a request to generate traffic. That traffic is sent through the communication pipes to the simulator.

– If a request comes to analyze traffic (rising edge of an analyzer clock), the traffic is retrieved from the simulator through the communication pipes. The correct analyzer

tree is requested to analyze this traffic.

– If there is a request for a VCI action, that request is processed. The CycleFunction is called before processing this VCI request.

When the first two cases finish, the main loop scheduler resumes watching the communication pipes. In the third case, the main loop scheduler finishes and the test plan execution is resumed.

4. Test plan execution is resumed until another WAIT_CYCLE is encountered. Scheduling continues with step 2. Note that a VCI command in the testprogram is also treated as a WAIT_CYCLE.

Before the generator or analyzer function is called that should generate or analyze traffic, all CycleObjects attached to that generator or analyzer are executed. This happens in both standalone and cosimulation mode.

Fig. 15 shows the scheduling in standalone mode. Fig. 16 shows the scheduling in cosimulation mode.

An important aspect of the present invention is to be able to view framed traffic data in a convenient way. Hence, the present invention includes a user graphical interface to view framed data. It displays the frames generated and/or analyzed by the verification or test bench system, with a possibility to traverse through the hierarchy of frames.

The viewer has two parts. One part is inside the verification or test bench system as is used to prepare data in such a way as to make viewing more robust. It collects information on the traffic while it is being generated or analyzed and annotates it. The other part is a viewer application. The additional information associated with the traffic data added by the first part is used by the viewer part to view the data and navigate through it. The viewer displays the information collected in the first part. In the test bench system, traffic data is divided into pieces, e.g. for ATM traffic, the verification or test bench system does not generate a complete cell at a time, but a single byte from a cell each time. Together with each piece of traffic, not only the value of the traffic (the value of the byte in the ATM cell) is stored, but also some extra parameters or descriptors of the data like start-of-cell indication, end-of-cell-indication. This piece of information travels through the whole hierarchy of blocks and block communication means.

The viewer part inside the test bench system adds some more information. For

each block a piece of traffic passes through, information about the current state of the block and the current state of the piece of traffic is added to the piece of traffic. As a piece of traffic travels through the hierarchy of blocks, all required viewer information is created. The descriptor information is preferably added at the block communication means. The file should be dumped before the data stream leaves the generators so that the additional information does not disturb the operation of the simulated device. Fig. 17 shows how viewer information is gathered in the generators. Fig. 18 shows the same thing for analyzers.

Information that can be added per level is:

- value at the current level in the hierarchy (value can change due to scrambling, ...)
- The tag of the block
- byte and frame counter (as provided by the common interface for all blocks)
- payload, overhead, start-of-frame, end-of-frame, head-of-packet, tail-of-packet indications (also provided by the common interface for all blocks)
- amount of traffic (number of bits)
- scrambled/descrambled version of the piece of traffic is applicable
- For analyzers, indication on how the piece of traffic was analyzed. Indication available : incorrect value, not aligned, not checked, piece of traffic was scrambled/descrambled, no indication (meaning correct value). These indication have to be explicitly set by the analyzer.

At the end of the tree, all the viewer information can be dumped in a file. The end of the tree is the node in the tree where the piece of traffic passes last. For generators, this is the top of the tree, for analyzers this is in the leaves of the tree. It's also possible to dump the information in the file at an intermediate level. This is used when the particular piece of traffic is overhead at a certain level and is not passed to the next analyzer.

The contents of the dumped file can be viewed with the viewer. The file created by the dump for the viewer contains all information to be able to display each frame generated or analyzed by the test bench system and to navigate between them, each frame, meaning not only the frame at the top of the tree, but also in intermediate levels. When a frame at an intermediate level is displayed, all information/traffic belonging to frames higher up in the hierarchy are hidden. Only the information/traffic for the selected level is shown. The user can navigate up and down through the

hierarchical layers of data as in a file structure.

In addition extra indications a byte has can be displayed, e.g. using code such as a combination of color codes, underlining, strike through, bold font, adding a box, etc. For example, if a bit belongs to overhead in a frame it can be displayed in red, payload in blue. Traffic that is neither overhead, nor payload is displayed in green. Traffic that the test bench system has analyzed as being incorrect, may have a cross over the value.

The viewer optionally has the following functions:

- Search for a specific type of traffic data (overhead, payload, errored traffic).
- A built-in distance calculator to calculate the distance, in number of bytes, between two bytes.
- The values of the bytes/bits at a specific level can be dumped to a text file.
- When the test bench system is still running, the viewer can automatically update itself whenever the dumpfile changes.
- Highlight (draw a box around every byte/bit) any level when viewing another level.
- Set a marker on a byte. The location of the marker is remembered when browsing through the hierarchy.

Fig. 19 and 20 show some screenshots of the viewer. The screen of Fig. 20a is from the same dump file as Fig. 19 but at a lower level in the hierarchy. Actually, the level shown in the screenshot of Fig. 20a is the same level that is highlighted in the screenshot of Fig. 19. In operation a control element of the user graphic interface is selected, e.g. by using a pointing device such as a mouse or by keyboard input or similar. Selection of this control element sends a signal to the verification system program running on the computer to display the hierarchical structure of the data preferably in a new and separate structure window which can be tiled or cascaded with a window to display the data. The verification system program make use of the fact that the framed data was generated under its control when the data stream was generated. Therefore with knowledge of the protocols involved and the behavior of the simulated device, the verifications system program reconstructs the hierarchical tree structure of the analyzed data and displays it. This display is conveniently in the form of a tree structure in the structure window. By selecting a part of this tree, e.g. by mouse double clicking or pressing "enter" on the keyboard, a signal is sent to the verification program to collect and display the framed data associated with that part of

the hierarchy. At the same time the displayed data is annotated with one or more of the codes given above, e.g. the start bit of a frame may be displayed with a recognizable code. In particular the framed data is displayed in columns and rows. An example, of the detail displayed by the user graphical interface in accordance with the present invention is shown in Fig. 20b, in which column number, frame number, a marker, byte reference at cursor position, marker information, input data file and the structure window can be determined.

Fig. 21 is a schematic representation of a computing system which can be utilized with the methods and in a system according to the present invention. A computer 41 is depicted which may include a video display terminal 14, a data input means such as a keyboard 16, and a graphic user interface indicating means such as a mouse 18 for use as a pointing device with the graphical user interface in accordance with the present invention. Computer 41 may be implemented as a general purpose computer, e.g. a UNIX workstation, a personal computer.

Computer 41 includes a Central Processing Unit ("CPU") 15, such as a conventional microprocessor of which a Pentium III processor supplied by Intel Corp. USA is only an example, and a number of other units interconnected via system bus 22. The computer 41 includes at least one memory. Memory may include any of a variety of data storage devices known to the skilled person such as random-access memory ("RAM"), read-only memory ("ROM"), non-volatile read/write memory such as a hard disc as known to the skilled person. For example, computer 41 may further include random-access memory ("RAM") 24, read-only memory ("ROM") 26, as well as an optional display adapter 27 for connecting system bus 22 to an optional video display terminal 14, and an optional input/output (I/O) adapter 29 for connecting peripheral devices (e.g., disk and tape drives 23) to system bus 22. Video display terminal 14 can be the visual output of computer 10, which can be any suitable display device such as a CRT-based video display well-known in the art of computer hardware. However, with a portable or notebook-based computer, video display terminal 14 can be replaced with a LCD-based or a gas plasma-based flat-panel display. Computer 41 further includes user interface adapter 19 for connecting a keyboard 16, mouse 18, optional speaker 36. Data may be input to the computer 41 from an external telecommunications device 20 such as personal computer or work station via a network 40 such as the Internet. This allows transmission of descriptor

files comprising a representation of a telecommunications device be simulated over a telecommunications network 40, e.g. entering a description of a physical system at a near location and transmitting it to a remote location, e.g. via the Internet, where a processor carries out a method in accordance with the present invention and returns a parameter relating to the physical system to a near location.

Computer 41 also includes a graphical user interface that resides within machine-readable media to direct the operation of computer 10. Any suitable machine-readable media may retain the graphical user interface, such as a random access memory (RAM) 24, a read-only memory (ROM) 26, a magnetic diskette, magnetic tape, or optical disk (the last three being located in disk and tape drives 23). Any suitable operating system and associated graphical user interface (e.g., Microsoft Windows) may direct CPU 15. In addition, computer 41 includes a control program 51 which resides within computer memory storage 52. Control program 51 contains instructions that when executed on CPU 15 carry out the operations described with respect to any of the methods of the present invention.

Those skilled in the art will appreciate that the hardware represented in FIG. 21 may vary for specific applications. For example, other peripheral devices such as optical disk media, audio adapters, or chip programming devices, such as PAL or EPROM programming devices well-known in the art of computer hardware, and the like may be utilized in addition to or in place of the hardware already described.
In the example depicted in Fig. 21, the computer program product (i.e. control program 51) can reside in computer storage 52. However, it is important that those skilled in the art will appreciate that the mechanisms of the present invention are capable of being distributed as a program product in a variety of forms, and that the present invention applies equally regardless of the particular type of signal bearing media used to actually carry out the distribution. Examples of computer readable signal bearing media include: recordable type media such as floppy disks and CD ROMs and transmission type media such as digital and analogue communication links.

The present invention has been exemplified in a computer based verification system known as CimTester™. In the annex a tutorial relating to this product provides further explanation with respect to the system and to the use of the graphics interface described above and referred to in the tutorial under the name FrameViewer™.

Annex

1 About This Tutorial

CimTester is a software tool for generating and analyzing data traffic of various kinds of communication protocols (SONET, SDH, ATM, PDH, ...). CimTester is constructed with primitive elements, called blocks. There are generator blocks, that do the mapping operations, and analyzer blocks, that do the demapping operations. Using simple configuration files, one can describe the stream hierarchy. Test programs to configure the blocks are written in C++.

By means of some examples, this tutorial takes you step by step through the process of describing a generator for a simple data stream. This data stream will be displayed in the FrameViewer™. Then you will learn to set up analyzers for the generated stream, and finally you will learn to configure the CimTester according to a user-specified test program.

This chapter contains a summary section with a brief description of each chapter, and a Documentation Conventions section.

1.1. Audience

The tutorial is targeted for engineers that want to create test benches to verify their telecom designs.

Participants should have:

- Some experience with Unix.
- Some exposure to the C and C++ language

- Some knowledge about data communication protocols, like SONET, SDH, PDH, ...
- Some knowledge about the CimTester architecture (see slide show)

1.2. Tutorial summary

This tutorial will briefly demonstrate some of the powerful key features of the CimTester. It is organized as follows:

- Exercise 1: Generating a simple data frame – explains how you can make the CimTester generate a basic data stream
- Exercise 2: Getting acquainted to the FrameViewer™ – in this exercise you will learn to use the FrameViewer™ to see the generated data frames
- Exercise 3: Adding framing hierarchy – you will construct a slightly more complicated example by using a more complicated framing hierarchy. You will learn to use more powerful features of the CimTester and FrameViewer™
- Exercise 4: The analyzers at work – until now you only generated frames. In this chapter you will add analyzers to the CimTester configuration files and learn how to interpret analyzer data in the FrameViewer™
- Exercise 5: Specifying a test program – explains how you can have full control over the behavior of both generator and analyzer blocks. Feel the power of CimTester!

For a deeper description of the details and configuration options of the CimTester, you will have to take a look at the user manual. For the brave programmers who will add their own protocols and therefore make CimTester an even more complete tool, a developers tutorial is available.

1.3. Conventions

`monospace`	In examples, shows system prompts, text from files, error messages, and reports printed by the system. In text, used for commands, variables, class names, reserved words, filenames, and pathnames.
`monospace`	In examples, shows user input.
/	Indicates levels of directory structure.
\	Indicates a continuation of a command line.
name1→name2	Shows a menu selection. name1 is the menu name, and name2 is the item on the menu.

2 Generating a simple data frame

In this exercise, you will learn how to use the CimTester executable, what it's configuration files look like, and what startup options it has. As an example, we will generate VC-3 frames. You will also become familiar with the data generator block, a special block that generates payload data patterns according to a wide variety of algorithms.

2.1. The CimTester executable

The CimTester can be used either in co-simulation mode[1] or in stand-alone mode. In this tutorial, we will focus on the stand-alone mode. This stand-alone mode of the CimTester comes in the form of an executable which is called `CimTester`.

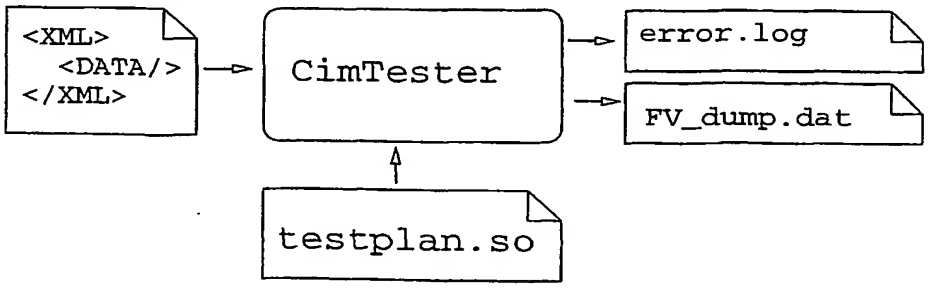


Figure 2.1: Inputs and Outputs for the CimTester

CimTester needs some input files specifying the desired framing hierarchy, some parameters and optionally a C++ test program. While it runs, it produces an error

[1] Co-simulation is available for the simulators Scirocco and Modelsim

log file, and optionally information for the FrameViewer™, a tool that visualises how the CimTester composed or decomposed a data stream.
The `CimTester` executable takes several options. For now, we will focus on three options to configure the frame generation path of the CimTester:

`-g filename.xml`	Read the framing hierarchy from `filename`
`-G dumpfile.dat`	Dump the trace file for the FrameViewer™ to `filename`
`-c limit`	Specify the number of cycles to run

2.2. Specifying the generator framing structure

It's our goal to generate one or more VC-3 frames. To configure the CimTester for this operation, we have to write a small configuration file. The configuration files that specify the framing hierarchy are written in XML-style[2]. Specifying hierarchy in a XML file is natural, so it's a very suitable format. The configuration file used to specify the CimTester generator hierarchy should have one single XML parent. This parent specifies the framing structure of the single byte stream that will be generated by the CimTester.
For this first exercise, we will connect a *data generator*[3] to the VC-3 block. Take your favorite text editor and enter the following configuration file:

```
<!-- This is a comment that will be discarded  -->
<VC type="3">
  <data>
  </data>
</VC>
```

This file tells the CimTester that the data generated by the data generator must be encapsulated in a VC-3 frame. Save this file as `generator.xml`. Please be sure to respect case since the configuration file is case sensitive.

[2]Although you don't have to be a XML expert to work with CimTester, you can find a XML tutorial at `http://www.projectcool.com/developer/xmlz/`

[3]A data generator is a special generator block that only generates data, it has no children. You will also meet his brother, the data analyzer, later in this tutorial.

Note that a XML file always exist of a set of <TAG> ... </TAG> pairs. These pairs are nested to create the hierarchy of more complicated framing structures we will use in further examples in the tutorial.

2.3. The FrameViewer™ file

All bytes that travel through the CimTester generator and analyzer blocks get annotated with debugging information. The offset of the byte in the frame, whether it was payload or overhead, whether a Start Of Frame signal was generated, and much more is remembered for every framing block that a byte passes. When the byte leaves the CimTester, this information is dumped to a file. A special program, called FrameViewer™, will interpret this dump file and give you a graphical representation of how the CimTester composed or decomposed its frames.
Bytes can leave the CimTester at two locations: either at the single output of a generator chain (the point marked with 1 in figure 2.2), or, for analyzers, at all levels where overhead bytes are dropped and when the final data leaves the system (the points marked with 2 in figure 2.2). The analyzer blocks and the generator blocks dump their trace data in different files. The names of these files are specified by command line options.
CimTester will by default not generate these dump files, since it will run considerably slower. You can turn on the generation of these dump files for the generators using the `-G` switch of the `CimTester` binary. In the remainder of this tutorial, dump files will always be created to allow visual inspection of the CimTester results.

2.4. Running the CimTester

After running a setup script[4] that sets some environment variables, type:

[4]this setup script should be delivered on the installation cd or tar file

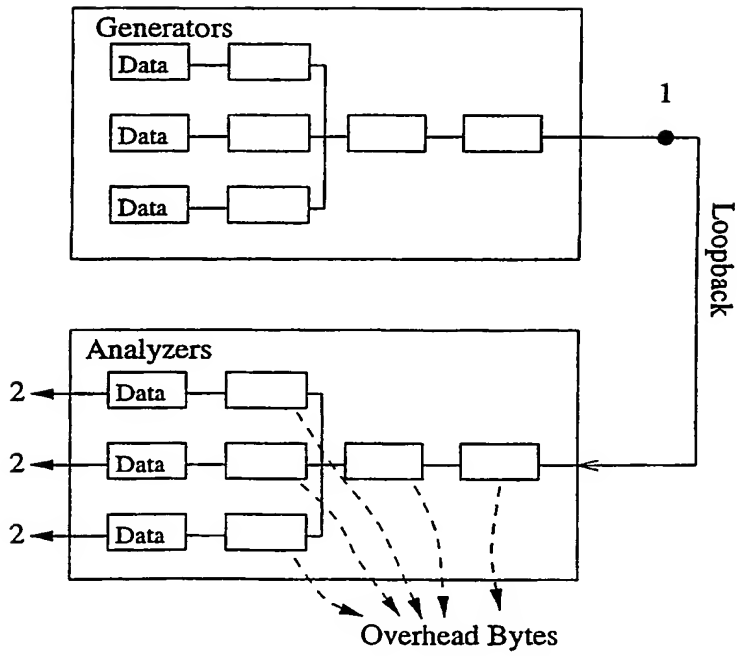


Figure 2.2: How data is sent to the dump files

```
>  CimTester -g generator.xml \
     -G generator.dat -c 5000
Running for 5000 cycles
>
```

This tells the CimTester to run for 5000 cycles, and generate frames according to the framing pattern found in the file `generator.xml`. The flow of the data through the CimTester is annotated to the bytes and dumped in the file `generator.dat`. Two new files will be generated:

- `generator.dat`: the FrameViewer™ dump file described above
- `error.log`: will hopefully be empty

Do not try to look at the contents of `generator.dat`, it is mostly binary crap. You will learn to work with the FrameViewer™ in a jiffie.

3 Using The FrameViewer™

Visual inspection of nicely formatted data frames often gives a designer the opportunity to spot a bug or the reason of unexpected behavior faster than staring at a waveform display. While CimTester gives designers the possibility to check the byte stream integrity by means of generator and analyzer error log files and inspection of the resulting byte streams, it also introduces a powerful visual verification tool: the FrameViewer™.

3.1. FrameViewer™ Basics

The best way to get familiar with the FrameViewer™ is probably to use it. Before you start however, it is important to realize that the FrameViewer™ will not interpret a byte stream, but will display how the CimTester has generated or interpreted the bytes.

The FrameViewer™ uses colors, font styles, status bars, rectangles, ... to explain to the user what annotation information it has found in the dump files. In this chapter we will explain step by step what all these exciting new colors and symbols mean.

3.2. A Session with the FrameViewer™

Start the FrameViewer™ by typing

```
> FrameViewer
```

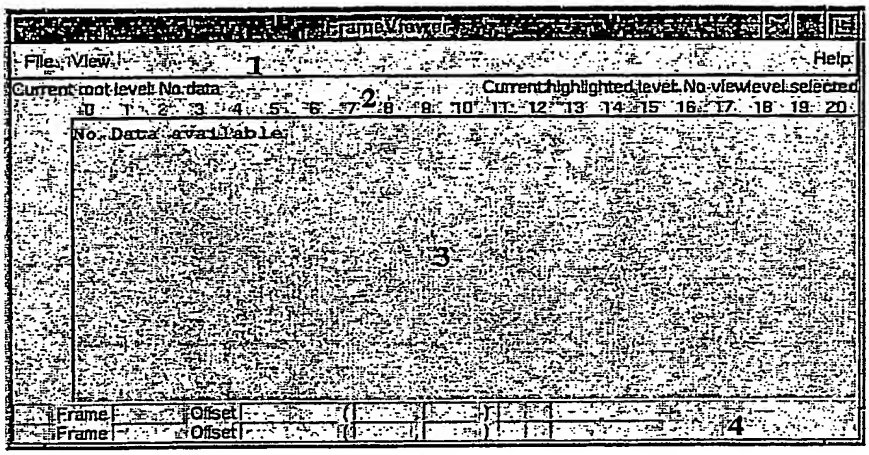


Figure 3.1: The empty FrameViewer™

at your shell prompt. A mostly empty window that looks like figure 3.1 will pop up. In this window, you see (from top to bottom) a menu bar (1), a text line telling you where you are in the framing hierarchy (2), a big empty place where frame data will come (3), and two status bars (4): the top one will display information about the pointed byte, the bottom one will display information about the byte that a marker selects.

Select File→Open and pick the `generator.dat` that you created in the previous exercise. After successful loading, you will see blue bytes, red bytes and white XX'es[1]. Also watch the status bars while moving your mouse pointer: the top bar will always show you information about the byte at your mouse cursor. See figure 3.2 for an explanation of the status bar fields. You can select a byte by clicking on it. Information about this byte will be depicted at the bottom status bar, together with the distance to the pointed byte.

The colors of the bytes always indicate payload/overhead status :

- blue: The byte is a payload byte

[1] scroll to the bottom to find them

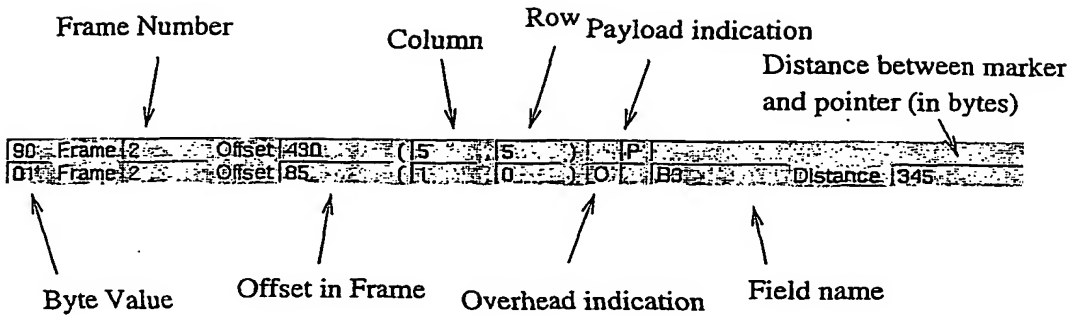


Figure 3.2: The FrameViewer™ status bars

- red: The byte is an overhead byte
- green: The byte is no overhead and no payload
- pink: The byte is both overhead and payload

Colors 'green' and 'pink' are special cases; we will come back to them when we meet them later in the tutorial.

When data is missing from a frame, two white X's will appear instead of the hexadecimal value of the byte. You will typically see these X's at the end of your file since your last frame will probably be incomplete. We will see later that these X'es also appear in analyzer streams, for example when frame synchronization is lost.

In the File→Write Frames menu, you can dump the frames to a text file. Select the range of the frames to dump, indicate whether you want to dump whole frames or just some bytes of them (e.g. to see the pointer justification bytes change over a range of frames). When you're dumping whole frames, it might be interesting to dump them 'vertical': the frames will be written to the file rotated 90 degrees clockwise, which gives a far more easier format to print your frames, especially when they are long. With the 'Be Verbose' box selected, frame and line counters will be exported to facilitate interpretation of the file. Try to dump a frame with the 'Be Verbose' and the 'Dump Vertical' options enabled to a file called `hexout.dump`

The View→Structure menu pops up a window that displays the hierarchical framing structure. In our case, it does not look very impressive: the top level is VC-3, and there is a single level below it, which produces data. Clicking on a byte in the FrameViewer™ window will show you it's origin in the structure window. In this case, all payload bytes' origin will be data, and all overhead bytes' origin will be added by the VC-3 block. The Structure window gets more interesting when we have a more complex framing structure as will be demonstrated in the next exercise.

4 Adding Framing Hierarchy

So far, we used an example with only one framing level – a VC-3 frame containing data – to demonstrate the most basic capabilities of the CimTester and the FrameViewer™. It's time now to make things a little more complex. In this section, we will add more framing hierarchy to the example and look how the FrameViewer™ can be used as a powerful tool to navigate through this hierarchy. As you can see in figure 4.1, we will encapsulate the VC-3 stream from the previ-

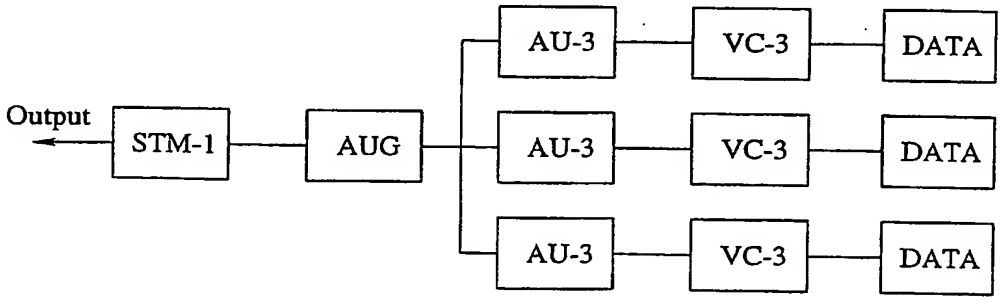


Figure 4.1: A more complex framing structure

ous example in a AU-3 frame. Three of these AU-3 frames are multiplexed in one AUG which is in turn framed in a STM-1 frame.

4.1. A More Complex Configuration File

Writing the configuration file for this framing structure is a little more complicated. If you use correctly indented XML, the framing hierarchy becomes clearer in the configuration file. Please notice that we used the `duplicate = "3"` option

to indicate that the AU-3 → VC-3 → DATA block is to be repeated three times. We could as well have written the substructure three times.

```
<!-- The base node of our framing hierarchy is STM-1 -->
<STM type="1">
  <!-- The STM-1 generator gets its input from a
       single AUG generator -->
  <AUG>
    <!-- The AUG frame is composed of three
         multiplexed AU-3 streams -->
    <AU type="3" duplicate="3">
      <!-- Every VC-3 is wrapped in a AU-3 -->
      <VC type="3">
        <!-- The VC-3 frames contain data -->
        <data>
        </data>
      </VC>
    </AU>
  </AUG>
</STM>
```

Run the CimTester again exactly like we did last time:

```
> CimTester -g generator.xml \
    -G generator.dat -c 5000
Running for 5000 cycles
>
```

And once again we are ready to see the results in the FrameViewer™.

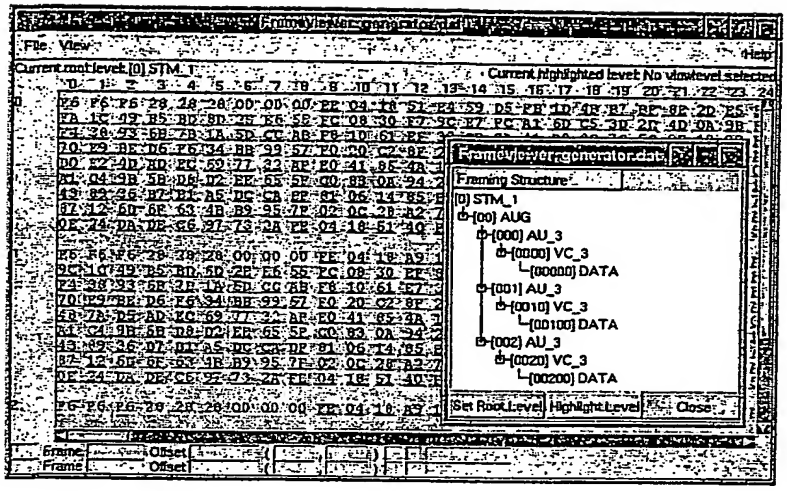


Figure 4.2: FrameViewer™ and the View→Structure window

4.2. Hierarchy Navigation In The FrameViewer™

Start the FrameViewer™:

```
> FrameViewer generator.dat
```

You may notice that several bytes are underlined. Underlined bytes are bytes that have been scrambled by the generator. FrameViewer™ gives you the possibility to descramble scrambled bytes by selecting View→Descramble from the top menu bar. The underlining will disappear when the bytes are descrambled. Select View→Descramble again to re-enable scrambling.

Select View→Structure in the top menu bar to see the framing structure of the file. The window that appears (figure 4.2) is a representation of the blocks that the CimTester selected to generate the stream in the dump file. The structure is not always exactly the same as what you entered in the configuration file `generator.xml`: sometimes the CimTester will split up mapper or demapper blocks in some smaller components. The user manual explains when this happens.

The number sequence in the square brackets that appears in front of the mapper name in the structure window is an important navigation number in the CimTester.

Every single mapper or demapper block in the CimTester can be unambiguously defined by such a string. The numbering scheme is simple:

- The top level block is numbered [00]
- The ID of all other blocks is composed as follows:
  - take the ID of the parent, and add a dot '.'
  - add the two-digit hex number that is the sequence number of the block at his level.

Keep the structure window open and click on the first byte of the first frame. You will notice that the STM-1 block will be highlighted in the structure window. This means that the selected byte was generated by the STM mapper. If you select the byte at row 3[1], column 0 of the first frame, the first multiplexed AU-3 stream is highlighted. Again, this means that the selected byte is overhead generated by the AU-3 mapper block in the first multiplexed sub-stream. If you click on the byte at position (4, 19), the generator block at path [00.00.01.00.00] will highlight. As a consequence, we can conclude that this is a true data byte, which is part of the second multiplexed data-stream in the frame.

Although the abovementioned methodologie gives you the possibility to find out where sub-frames are located within frames, the FrameViewer™ also offers a much more powerful way to see framing structure. Simply select AUG from the structure view and click the Highlight Level button. The FrameViewer™ now highlights all bytes that traveled through or were added by the AUG mapper. You can now see blue rectangles around a lot of bytes, and rounded rectangles around some other bytes. The rounded rectangles are bytes where the offset in the frame in 0 at the selected level; they can therefore be considered as the indication where a sub-frame starts. The rounded rectangles will turn out to be extremely helpful to track frame boundaries when pointer justification happens.

Now select a AU-3 stream and click the Highlight Level button. Some blue squares will disappear, some red squares will appear. It is important to know

[1]Remember that row, column and frame numbers always start at zero

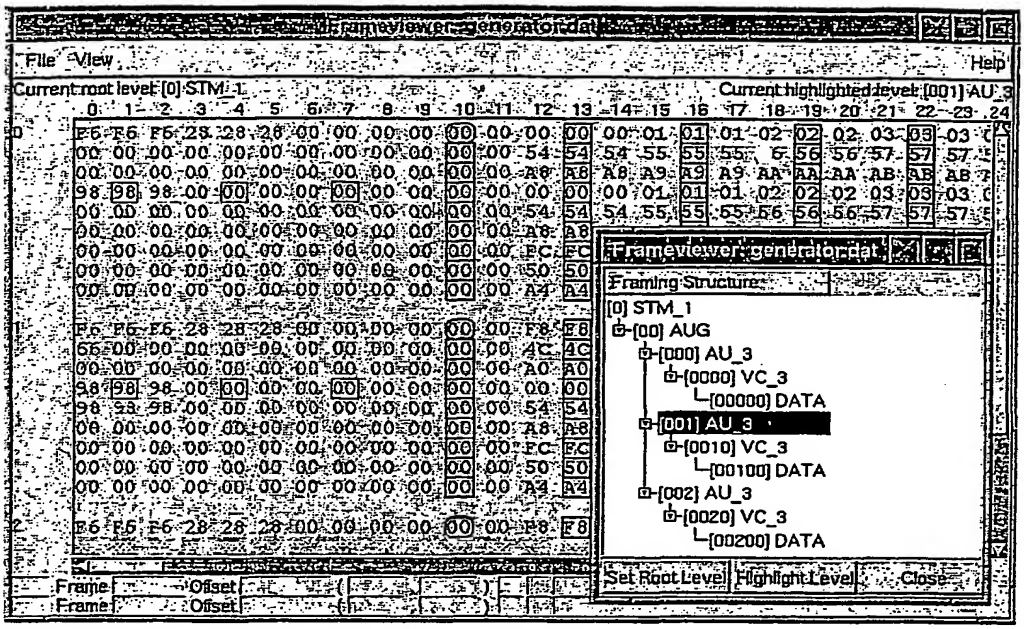


Figure 4.3: Same STM-1 frame, the second AU-3 stream is highlighted

that the color code for the (rounded) rectangles is exactly the same as for the bytes that you can see: blue = payload, red = overhead, ... So a red rectangle means that in the AU-3 level, this byte has been generated as overhead, while a blue rectangle means that the byte has been passed through as payload. Note that the currently selected view level is always displayed at the right top corner of the FrameViewer™.

The Set Root Level button allows you to descend in the frame hierarchy. If you press the button, the selected framing level will be extracted and displayed as if it were the resulting stream. Because people tend to get lost in all this framing levels, the root level is displayed at the top left corner in the FrameViewer™.

5 Using analyzers

Until now, we used the CimTester in a pretty dumb mode. All generated data was simply discarded at the end of the generation chain. Fortunately, we can capture the data from the generators and route it trough some analyzers. In this chapter you will learn how to dump the generated stream to a file, how to set the CimTester in loopback mode, and how to read a stream from a file to run it through the CimTester demapper blocks. That will enable you to make test vectors for your chips and analyze the data streams that your telecom chips produce. Note that it is not required to dump your streams to a file first; the CimTester can be used in a cosimulation mode.

5.1. Setting up CimTester for loopback mode

The loopback mode of the CimTester is configured by two new startup parameters:

`-a file` Specify where to find the framing hierarchy to analyze frames

`-A file` Specify where to dump the analyzer trace file for the FrameViewer™

Where the `-G` option dumps bytes at the end of the generation path, the `-A` option dumps bytes at all possible outputs of the analyzer path[1]. Likewise, all overhead bytes that are dropped by the demappers are also recorded in the analyzer dump file, so that a full graphical representation of the interpreted data stream can be displayed to the FrameViewer™ user.

[1] Remember figure 2.2

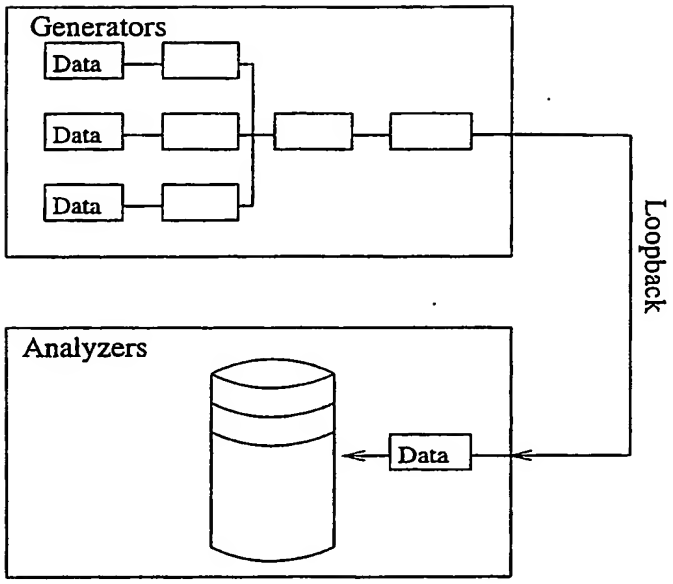


Figure 5.1: CimTester in loopback mode: dump to file

5.1.1. Dumping a data stream to a file

Just like we have a data generator, there also exist a data analyzer that checks the integrity of a data stream (PRBS, incremental counter, ...) and gives you the option to write the data to a file. Dumping a generated stream to a file is a specific configuration of a data analyzer: it only dumps the stream in a file. Figure 5.1 depicts a configuration where the generated stream is dumped in a file (CimTester is used to generate test vectors)

The generator configuration file might look like this:

```
<STM type="1">
  <AUG>
    <AU type="3" duplicate="3">
      <data>
      </data>
    </AU>
  </AUG>
```

```
</STM>
```

The analyzer configuration file has only one block:

```
<data file="output">
</data>
```

The `file="..."` option can also be used in the data generators: this will cause the generators to read data from a file[2].

5.1.2. Analyzing a data stream from a file

This is what you get if you swap the generator and configuration files mentioned above.

5.1.3. Loopback from generators to analyzers

In figure 5.2, we connected a set of generators back to the same set of analyzers. At first sight, one might expect that the data analyzers produce exactly the same data stream as the data generators have been generating, but that is not always the case: some frames are dropped because of out-of-frame alignment conditions etc. This mode of the CimTester is frequently used to test the CimTester itself. You can also use it to compose and verify your test plans.

5.2. Analyzer blocks in the FrameViewer™

Let's take the configuration file we used previously:

```
<!-- The base node of our framing hierarchy is STM-1 -->
<STM type="1">
  <!-- The STM-1 generator gets its input from a
       single AUG generator -->
```

[2]Don't try these configuration files: they will not work. File access must be enabled from the test program, which will be discussed in the next chapter

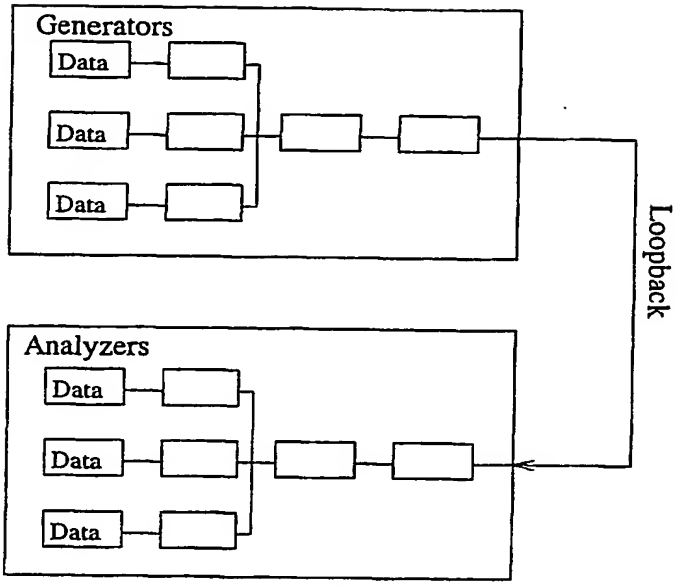


Figure 5.2: CimTester in loopback mode: analyze generated data

```
    <AUG>
      <!-- The AUG frame is composed of three
           multiplexed AU-3 streams -->
      <AU type="3" duplicate="3">
        <!-- Every VC-3 is wrapped in a AU-3 -->
        <VC type="3">
          <!-- The VC-3 frames contain data -->
          <data>
          </data>
        </VC>
      </AU>
    </AUG>
</STM>
```

And run the CimTester again:

```
> CimTester -g config.xml \
    -a config.xml -A analyzer.dat -c 10000
Running for 5000 cycles
>
```

We now generated a dump file for the analyzer path. Open it in the FrameViewer™ and you will notice at least two new FrameViewer™ features at once:

- The trashcan: All bytes that are dropped by an analyzer because there is no frame alignment are put together in a trashcan. The trashcan appears *after* the frame it belongs to. You can click on the thrashcan to see the out-of-sync bytes.

- Green bytes: Green bytes are bytes that have been indicated as not being payload and not being overhead. In general, green bytes are bytes of frames that are aligned, but which are not passed to the next framing level but are thrown away by the demapper. In this case, the STM standard specifies that the synchronization pattern should be encountered at least twice before payload is passed to the next framing level. Green bytes also appear in the ATM demapper, where the payload of idle cells is also thrown away.

Navigating through hierarchy by means of the View→Structure menu remains the same as for the generator blocks.
Some analyzer blocks tend to generate start-of-frame information for the analyzer blocks deeper in the hierarchy level. These generated start-of-frame signals can be visualized by selecting View → Show Generated SOF. Bytes in bold are passed to the next level with the Start of Frame signal asserted. If you highlighted some level, the generated start of frame bytes will have fat rectangles around them.
Errors encountered in the byte stream will be reported through the `error.log` file. The file could contain the following data:

```
Fr 0 By 0 (0, 0) Block DataAnalyzer[00000]: vc_.state.errors
Fr 0 By 0 (0, 0) Block DataAnalyzer[00100]: vc_.state.errors
Fr 0 By 0 (0, 0) Block DataAnalyzer[00200]: vc_.state.errors
```

This means that at frame 0, the data analyzer blocks with the given ID's have detected errors. Using a test program, you will be able to introduce much more interesting errors and warnings in the error log file.

6 Writing a test program

Now comes the best part. Controlling CimTester with a test program.

6.1. A very simple test program

We are going to use the same configuration files as in the previous example. We will configure a DataGenerator block.
Take the document containing the CimTester control signals. Look for the control signals for the `DataGenerator` block. There is a record `config` with a field called `mode`. This field determines how data is generated in a DataGenerator. The default value for this field is `INCREMENTAL`. We will change this to `DECREMENTAL`.
There are 3 DataGenerator blocks in the example. We still need to specify which block we want to configure. There are 2 functions defined that for this purpose. `mapG` and `mapA`. `mapG` is for generator blocks, `mapA` is for analyzer blocks. These functions transform the ID of a block into a number that can be used with the control signal records. This ID is the same one as you can see in the structural view of the FrameViewer™. You can also get the ID's of all the blocks by adding the command line switch '-v'
This line of code will do the trick :

```
VC_DataGenerator[mapG("00100", "data")].config.mode = DECREMENTAL;
```

The first argument is the ID of the block. The second argument is the name of the block. This second argument is for checking purposes only. If the block with the specified ID is not of the given type, an error message will be displayed.

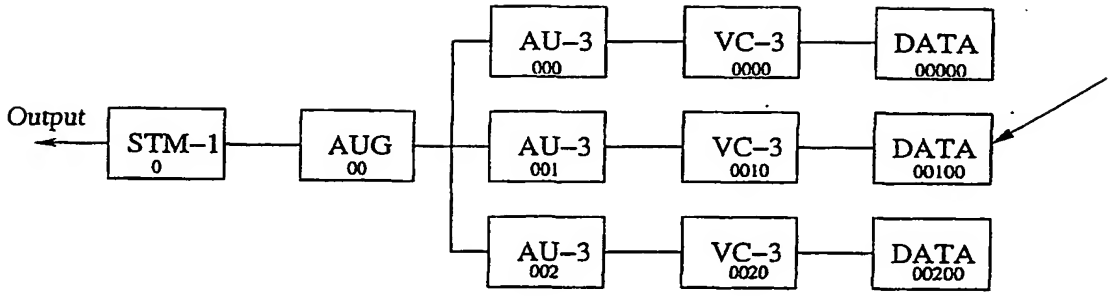


Figure 6.1: The DataGenerator block we are configuring

We now have to put some C sugar around our line of code. CimTester expects that our test program is in a function called CTestbench, so we put our line of code in a function with that name. Because our test program will eventually become a shared library that CimTester will load at the start of simulation, we need some extra C++ sugar around our function. This is a valid implementation of a test program testplan.C:

```
#include "VCI_CimTester.h"
#include "CimTester_SW.h"

extern "C" {
void CTestbench()
{
  VC_DataGenerator[mapG("00100", "data")].config.mode = DECREMENTAL;
}
}
```

The first 2 lines are always needed when writing a test program for CimTester. We now have a piece of C++-code which we need to compile[1]. The easiest way to compile it, is by letting cma generate a Makefile. You will need an init file for cma called cma.ini :

[1] You have to make sure you are using gcc-2.95.2 to compile the testplan. Other compilers will not work.

```
search_dir = $(CIMTESTER_HOME)
bin_dir = .
object_dir = objects
dependency_dir = objects
extra_compile_flags = -fPIC
extra_link_flags = -shared
exe_name_suffix = so
cxx_extension = C
```

Now you can run cma :

```
> cma testplan.C
```

Compile the testplan with :

```
> make -f cmaMakefile.testplan
```

Run the CimTester :

```
>  CimTester -g generator.xml -G generator.dat \
     -t testplan -c 5000
Running for 5000 cycles
>
```

When you open the FrameViewer™ , you should see that the data in the payload of the second VC3 is now decrementing.

6.2. Let the pointers move

Still using the same configuration files for the CimTester. We will only extend our test program. What we would like to do is, every eight frames increment the H1H2 pointer values of the first AU-3. This is how we do it.

```
#include "VCI_CimTester.h"
#include "CimTester_SW.h"

extern "C" {
void CTestbench()
{
  VC_DataGenerator[mapG("00100", "data")].config.mode = DECREMENTAL;

  VC_AUGenerator[mapG("000", "AU_3")].config.pointer_command = NORM;

  unsigned int frame_counter = 0;
  while (1)
    {
      WAIT_UNTIL(VC_AUGenerator[mapG("000", "AU_3")].state.frame_count !
                 frame_counter);
      frame_counter = VC_AUGenerator[mapG("000","AU_3")].state.frame_cou
      if (frame_counter % 8 == 0)
        VC_AUGenerator[mapG("000", "AU_3")].config.pointer_command = INC
    }
}
}
```

The WAIT_UNTIL command is a CimTester command that will stop the test program until the given condition becomes true.
Compile your test program and run it. Run it for a longer period now, otherwise you won't see any pointer movements.

```
>  CimTester -g generator.xml -G generator.dat \
     -t testplan2 -c 50000
Running for 50000 cycles
>
```

Open the FrameViewer™ again, open the structure view and set the highlight level at the first VC-3. Scroll down a bit and behold the pointer movement.

6.3. Let's play checkers

If you browse through the control signals document, you see a lot of control signal, and a lot of state signals, mainly reporting errors. When your simulation is running, you would have to check for all these signals and maybe display the errors on the screen in a human readable form.
This is were the checker come in. With the CimTester there are C++ classes which do it for you. They check the state signals and print out information in a human readable form when something goes wrong.
Suppose we would give a pointer increment every frame, after some time. An experienced SONET/SDH designer immediatly says : 'not allowed !'. Of course not, but we want to do some stress testing. This is the code :

```
#include "VCI_CimTester.h"
#include "CimTester_SW.h"

extern "C" {
void CTestbench()
{
  VC_DataGenerator[mapG("00100", "data")].config.mode = DECREMENTAL;

  VC_AUGenerator[mapG("000", "AU_3")].config.pointer_command = NORM;

  WAIT_UNTIL(VC_AUGenerator[mapG("000", "AU_3")].state.frame_count == 10);

  unsigned int frame_counter = 0;
  while (1)
    {
      WAIT_UNTIL(VC_AUGenerator[mapG("000", "AU_3")].state.frame_count !=
```

```
                frame_counter);
        frame_counter = VC_AUGenerator[mapG("000","AU_3")].state.frame_co
        VC_AUGenerator[mapG("000", "AU_3")].config.pointer_command = INC;
    }
}
}
```

Check in the FrameViewer™ that the CimTester is doing what we want.
Now we will add an analyzer to the CimTester. The analyzer has the same SDH structure so we can reuse the configuration file.
Compile the test program and run CimTester again.

```
> CimTester -g generator.xml -G generator.dat \
    -a generator.xml -t testplan3 -c 50000
Running for 50000 cycles
>
```

Look at the result with the FrameViewer™.
Now we are going to add the AUAnalyzerChecker.

```
#include "VCI_CimTester.h"
#include "CimTester_SW.h"
#include "AUAnalyzerChecker.h"

extern "C" {
void CTestbench()
{
  (void) new AUAnalyzerChecker(VC_AUAnalyzer[mapA("000", "AU_3")],
                               "AU : ", AUAnalyzerChecker::Verbose);

  VC_DataGenerator[mapG("00100", "data")].config.mode = DECREMENTAL;
```

```
    VC_AUGenerator[mapG("000", "AU_3")].config.pointer_command = NORM;

    WAIT_UNTIL(VC_AUGenerator[mapG("000", "AU_3")].state.frame_count == 10);

    unsigned int frame_counter = 0;
    while (1)
      {
        WAIT_UNTIL(VC_AUGenerator[mapG("000", "AU_3")].state.frame_count !=
                   frame_counter);
        frame_counter = VC_AUGenerator[mapG("000","AU_3")].state.frame_count;
        VC_AUGenerator[mapG("000", "AU_3")].config.pointer_command = INC;
      }
}

void CycleFunction()
{
  CheckerBase::cycleAllCheckers();
}

}
```

The CycleFunction is a function that is called at every cycle of the CimTester.
Run cma again, recompile and rerun the CimTester. Your screen should be full of nice error messages from the AUAnalyzerChecker.
There are all kinds of checkers. Look at the examples for more info.

Claims

1. Computer apparatus for displaying and manipulating sequences of frames of hierarchically organized framed data, comprising:

means for generating a graphical user interface on a display screen, the graphical user interface having means for displaying a representation of a hierarchy of the framed data,

a pointing device for selecting a portion of the representation of the hierarchy to generate a control signal, and

means responsive to the control signal to display a portion of the framed data corresponding to the selected portion of the representation of the hierarchy.

2. Computer apparatus according to claim 1, further comprising for coding data as displayed on the display screen as one of overhead, payload, neither overhead nor payload and incorrect.

3. Computer apparatus according to claim 1 or 2, further comprising at least one of: means for searching for a specific type of traffic data type, a distance calculator to calculate the distance, in number of bytes, between two bytes of data, means for dumping values of the bytes/bits at a specific level in the hierarchy, means for updating the display whenever a dump file changes, means for highlighting data on the display screen at any level when viewing another level, means for setting a marker on a byte of data, and means for maintaining the location of the marker when browsing through the hierarchy.

4. Computer apparatus according to any of the previous claims, further comprising a frame generator for receiving a frame sequence, the frame generator comprising means for processing the frame sequence in accordance with a protocol to form a modified frame sequence and for associating with each frame a description data structure for that frame, and

means for outputting the modified frame sequence and the frame description data structure.

5. Computer apparatus according to claim 4, further comprising: a frame analyzer for receiving a sequence of frames and a description data structure, the frame analyzer comprising means for separating the sequence of frames from the description data structure.

6. Computer apparatus according to claim 4 or 5, further comprising:

a first frame scheduler for triggering reading of an input frame sequence of the frame generator.

7. Computer apparatus according to claim 5 or 6, further comprising:

a second frame scheduler for triggering reading of an input frame sequence of the frame analyzer.

8. Computer apparatus according to any of claims 5 to 7, further comprising:

test program execution means for executing a pre-defined test program on the framed data by interacting with the frame generator and the frame analyzer.

9. Computer apparatus according to claim 8, wherein the test program execution means interacts with the frame generator and the frame analyzer through a set of interaction objects.

10. Computer apparatus according to any of the claims 5 to 9, wherein the framed data is input to the frame analyzer from a simulator or from a telecommunications device.

11. Computer apparatus according to any of claims 5 to 10, further comprising a plurality of frame generators and a plurality of frame analyzers, the pluralities being organized in a hierarchical tree structure.

12. A computer based verification system for the analysis and display of a sequence of frames of framed data, comprising:

means for generating a frame generator for receiving the frame sequence, the frame generator comprising means for processing the frame sequence in accordance with a protocol to form a modified frame sequence and for associating with each frame a description data structure for that frame, and

means for outputting the modified frame sequence and the frame description data structure.

13. The system according to claim 12, further comprising:

means for generating a frame analyzer for receiving a sequence of frames and description data structures, the frame analyzer comprising means for separating the sequence of frames from the description data structures.

14. The system according to claim 12 or 13, further comprising:

a first frame scheduler for triggering reading of an input frame sequence of the frame generator.

15. The system according to claim 13 or 14, further comprising:

a second frame scheduler for triggering reading of an input frame sequence of the frame analyzer.

16. The system according to any of claims 13 to 15, further comprising:

test program execution means for executing a pre-defined test program on the framed data by interacting with the frame generator and the frame analyzer.

17. The system according to claim 16, wherein the test program execution means interacts with the frame generator and the frame analyzer through a set of interaction objects.

18. The system according to any of the claims 13 to 17, wherein the framed data is input to the frame analyzer from a simulator or from a telecommunications device.

19. The system according to any of claims 13 to 18, further comprising a plurality of frame generators and a plurality of frame analyzers, the pluralities being organized in a hierarchical tree structure.

20. The system according to claim 19, wherein a communication means is provided between a generator and a next generator.

21. The system according to claim 19 or 20, wherein a communication means is provided between a generator and a next generator.

22. The system according to any of the claims 12 to 21, further comprising:
a graphical user interface for displaying the framed data annotated with at least part of the description data structure.

23. A method of analyzing and displaying a sequence of frames of framed data, comprising the steps of:
generating a frame sequence in accordance with a protocol,
associating with each frame a description data for that frame, and
outputting the frame and the frame description data.

24. The method according to claim 23, further comprising the steps of:

receiving a sequence of frames and a description data structure and separating the sequence of frames from the description data structure.

25. The method according to claim 23 or 24, further comprising inputting the framed data from a simulator or from a telecommunications device.

26. The method according to any of claims 23 to 25, further comprising:

displaying the framed data annotated with at least part of the description data structure.

27. A data carrier comprising a computer executable computer program for executing any of the methods of claims 23 to 26.

28. A computer program product for executing any of the methods of any of claims 23 to 26 when executed on a computing device.

29. A method of analyzing framed data comprising the steps of:

entering a descriptor file at a near location such as a terminal connected to a telecommunications network, the descriptor file comprising at least a representation of the operation of a telecommunications device;
transmitting the descriptor file over the telecommunications network to a remote location;
generating framed data at the remote location using a computer based system in accordance with any of the claims 1 to 11 or in accordance with a method in accordance with claims 12 to 15;
and receiving at a near location the modified framed data and description data associated with the frame data.

30. The method according to claim 29, further comprising storing the framed data and the description data at an intermediate node of the telecommunications.

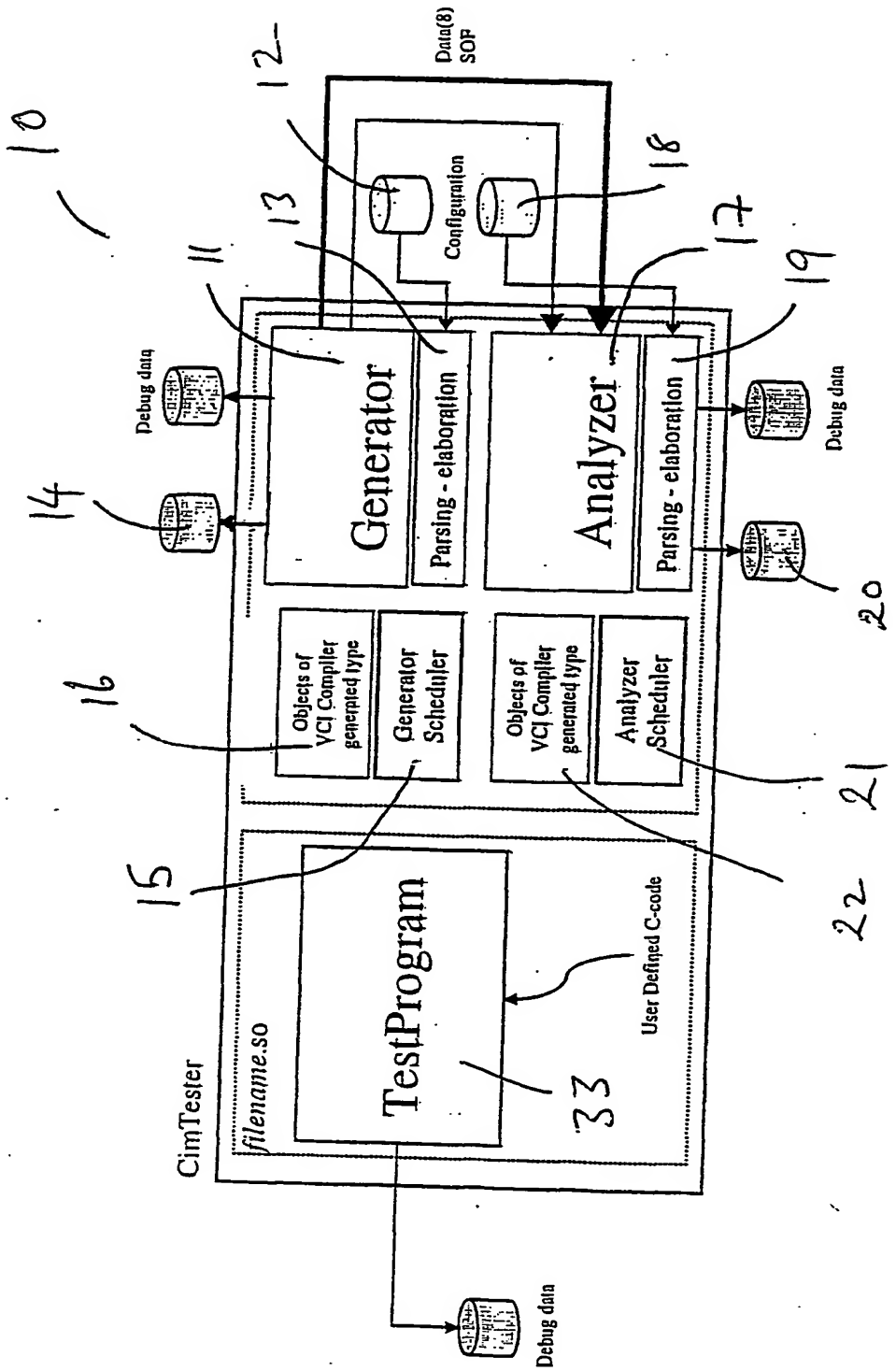


Fig. 1a

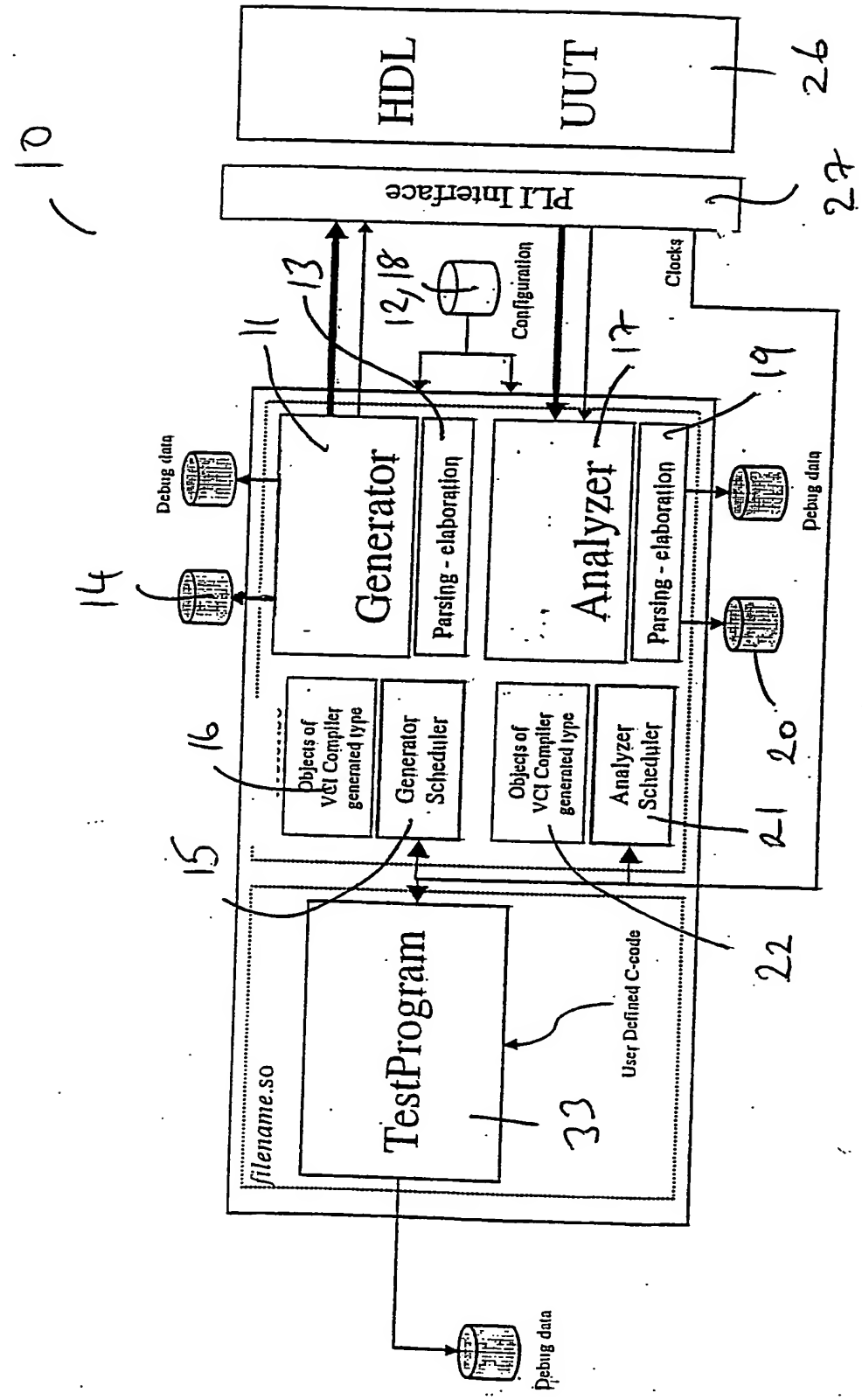


Fig. 1b

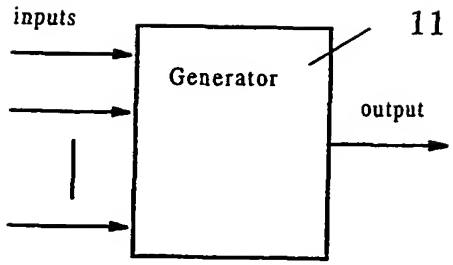


Figure 2

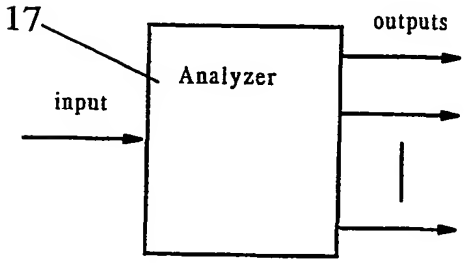


Figure 3

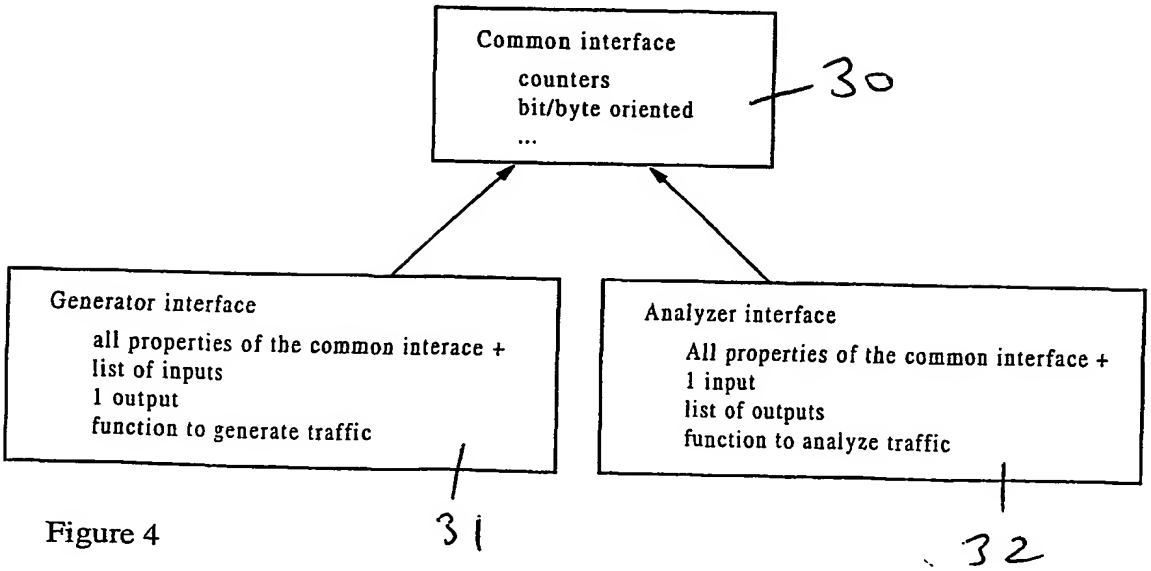


Figure 4

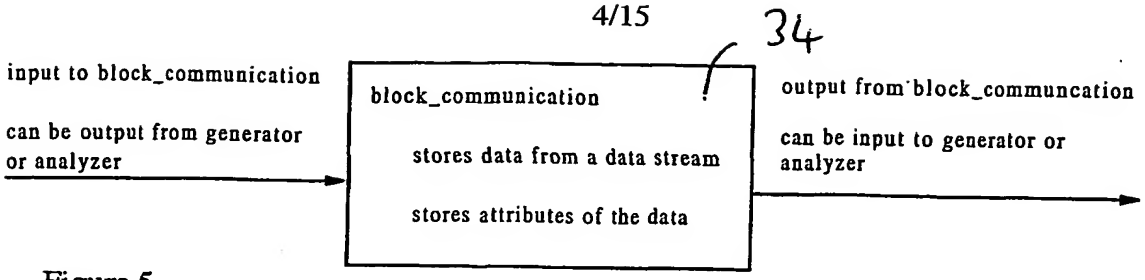


Figure 5

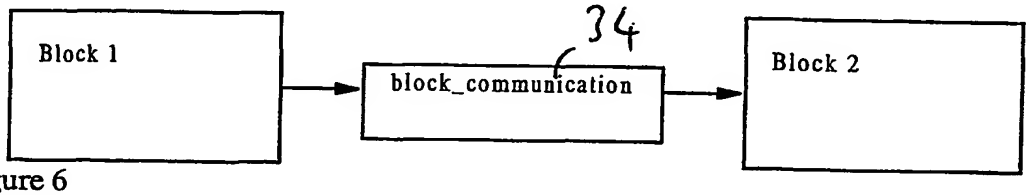


Figure 6

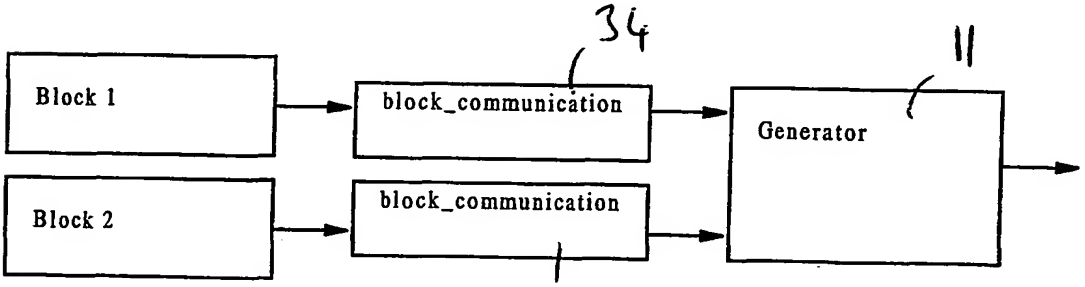


Figure 7a

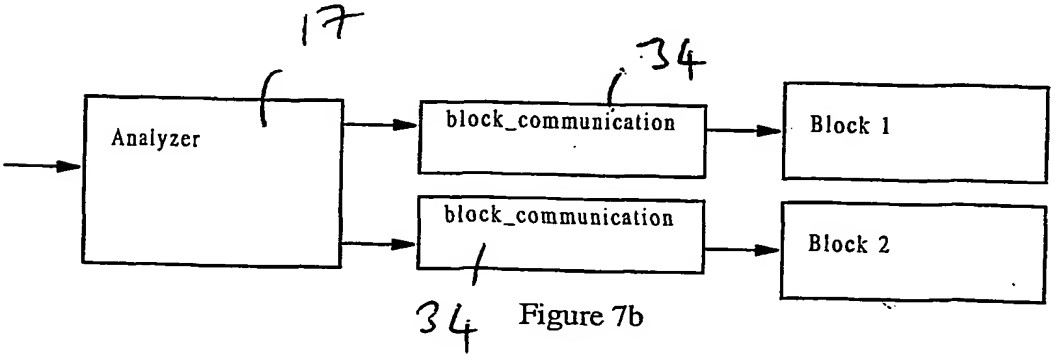


Figure 7b

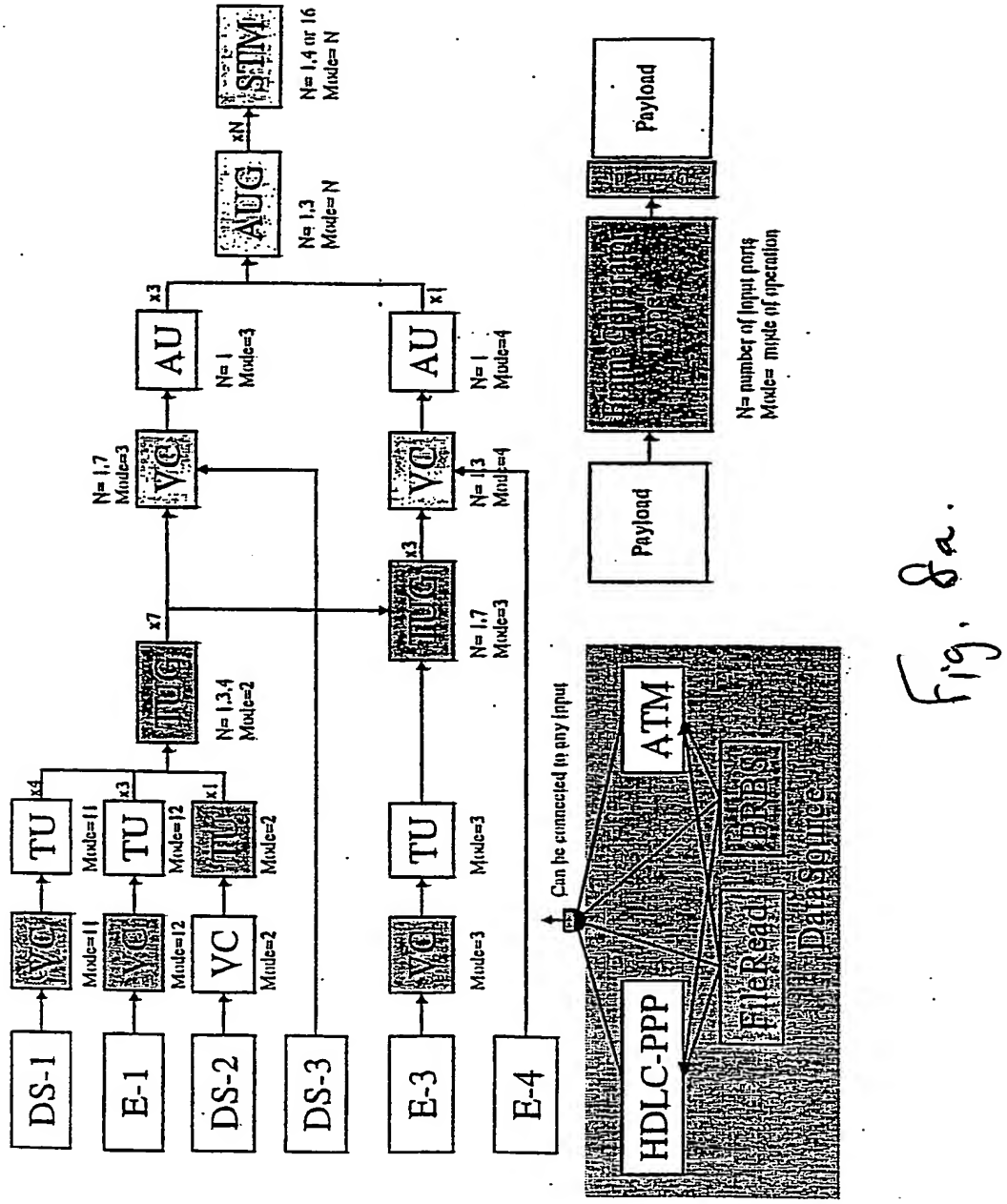


Fig. 8a.

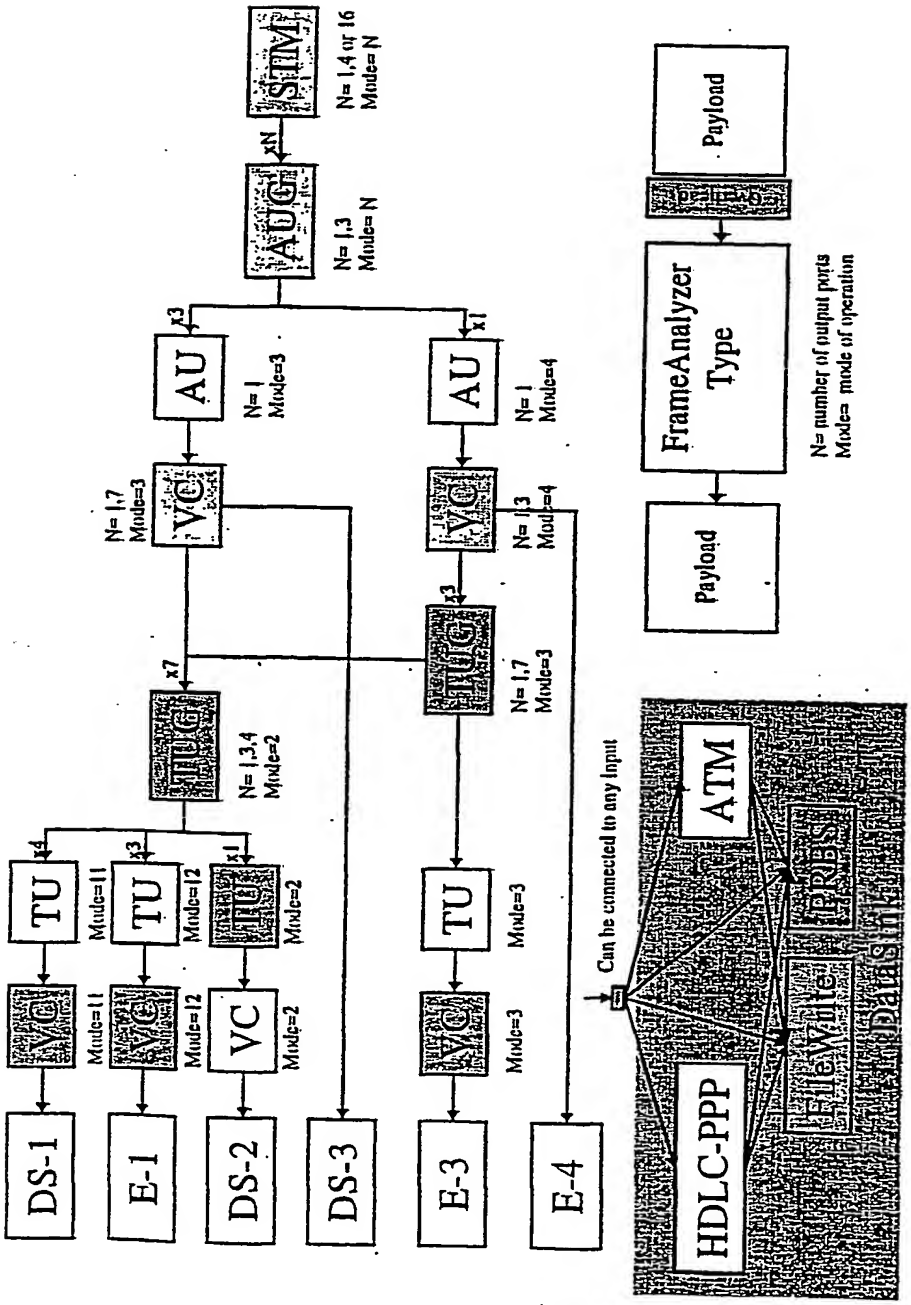


fig. 8b

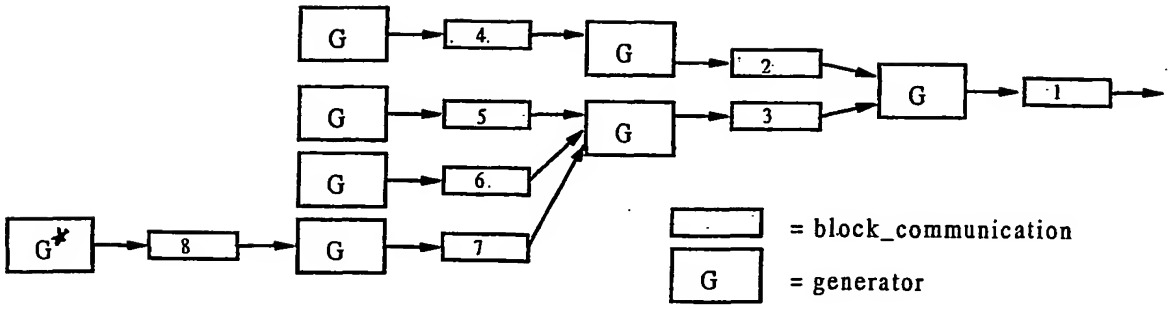


Figure 9

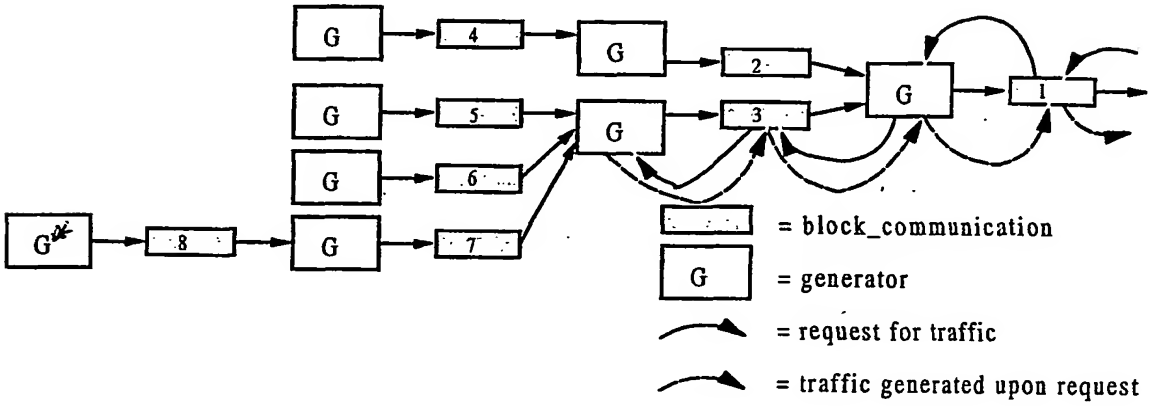


Figure 10

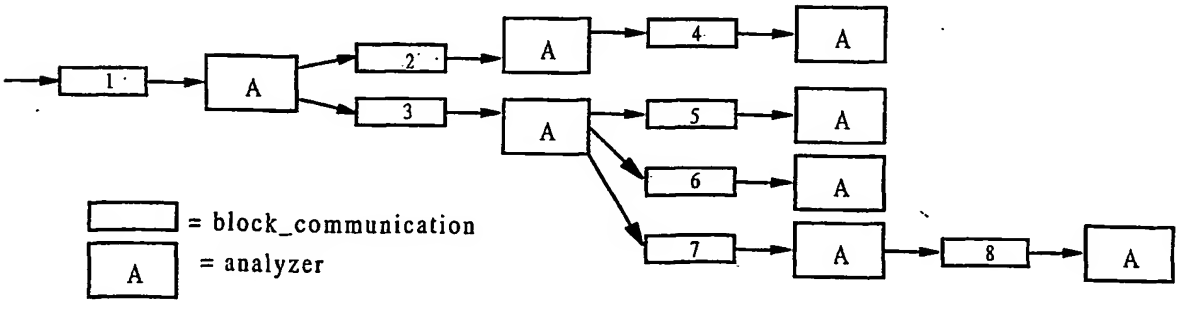


Figure 11

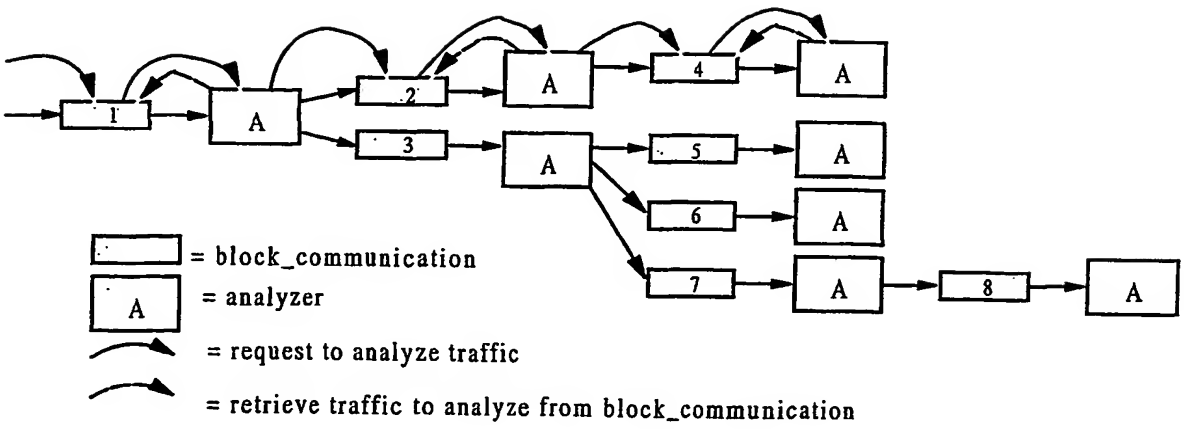


Figure 12

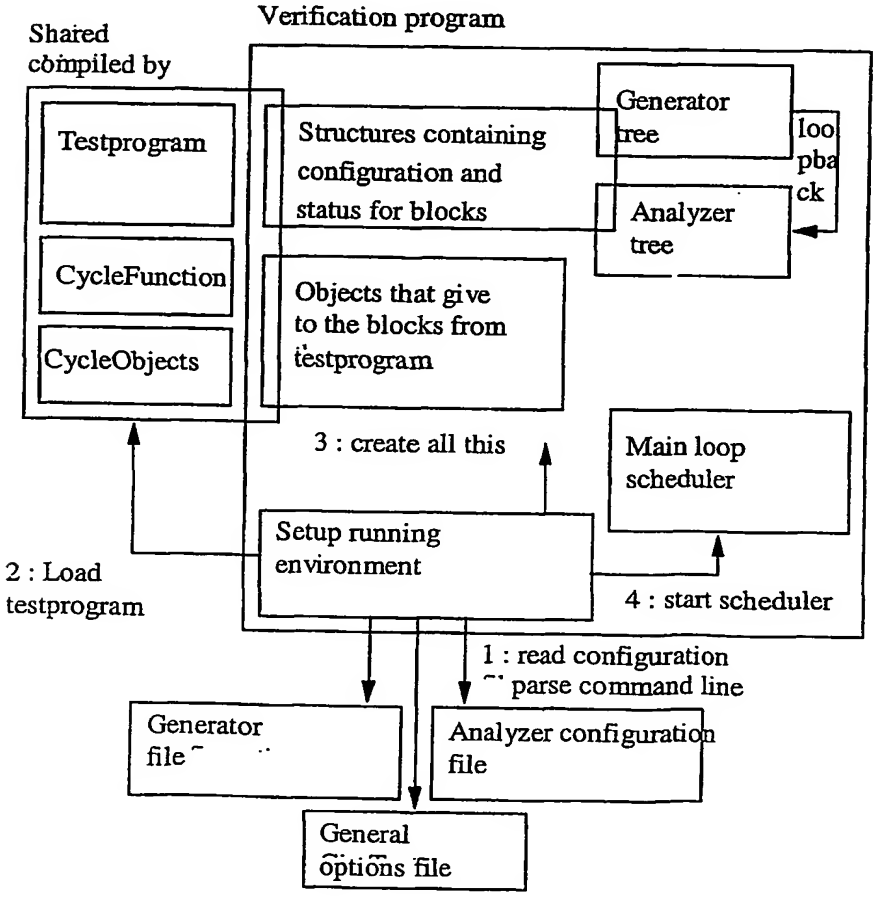


Figure 13

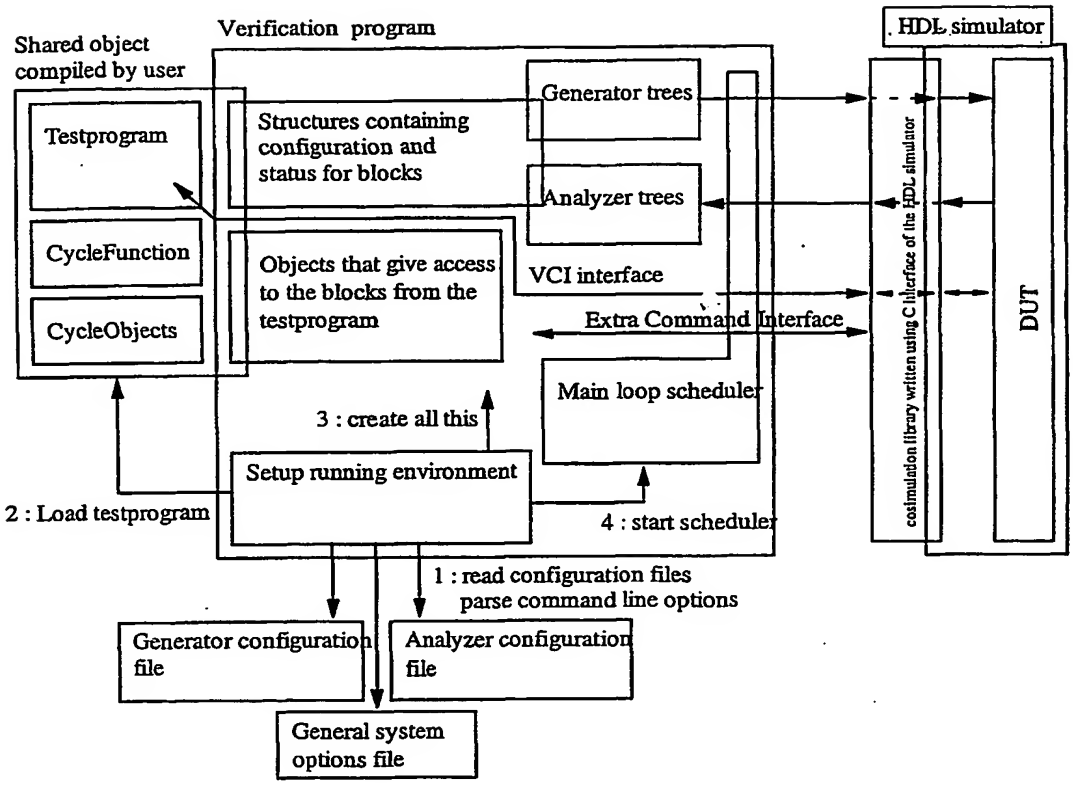


Figure 14

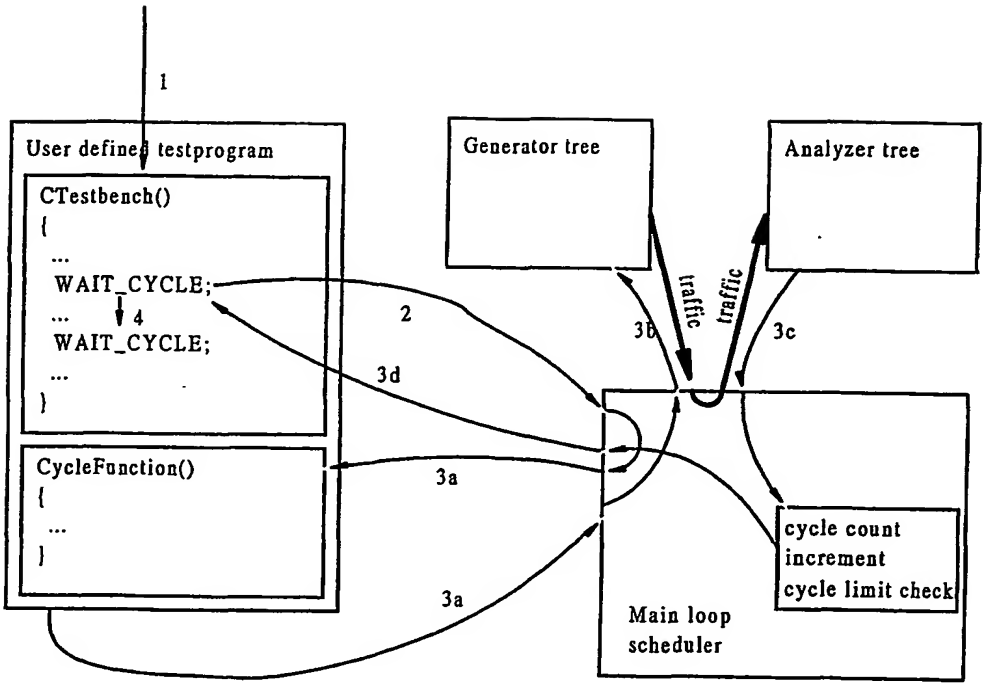


Figure 15

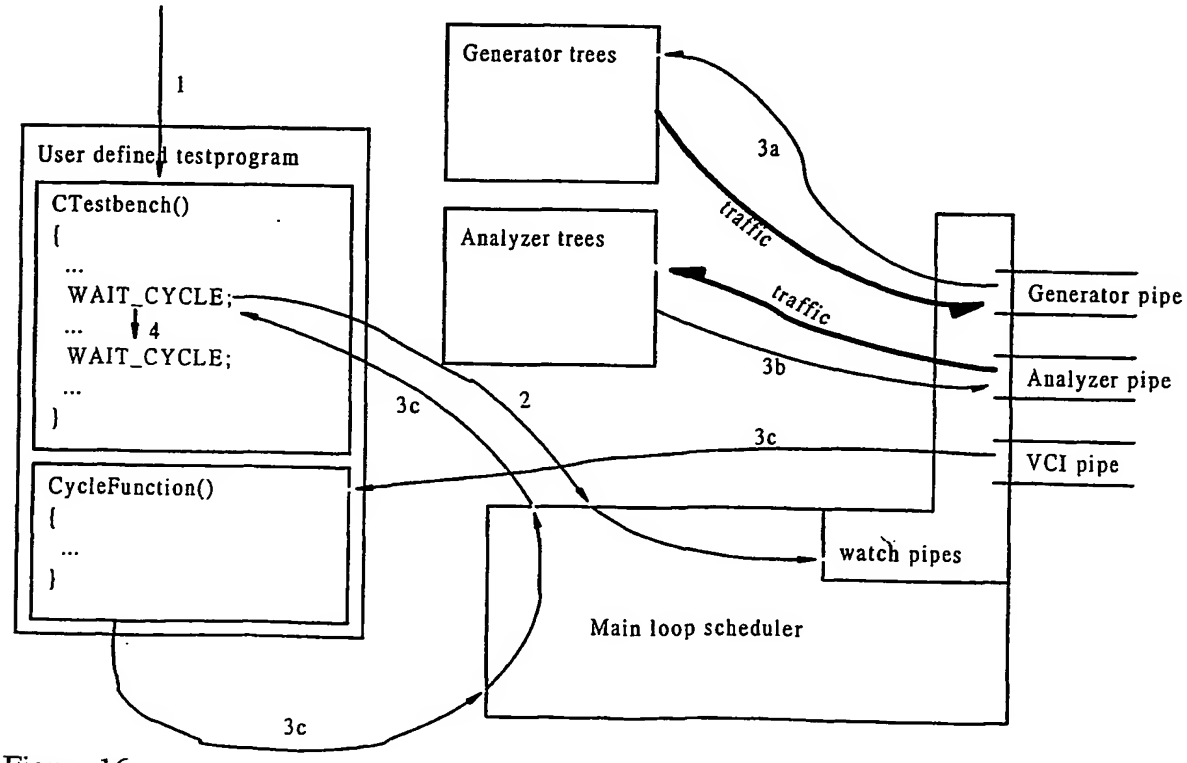


Figure 16

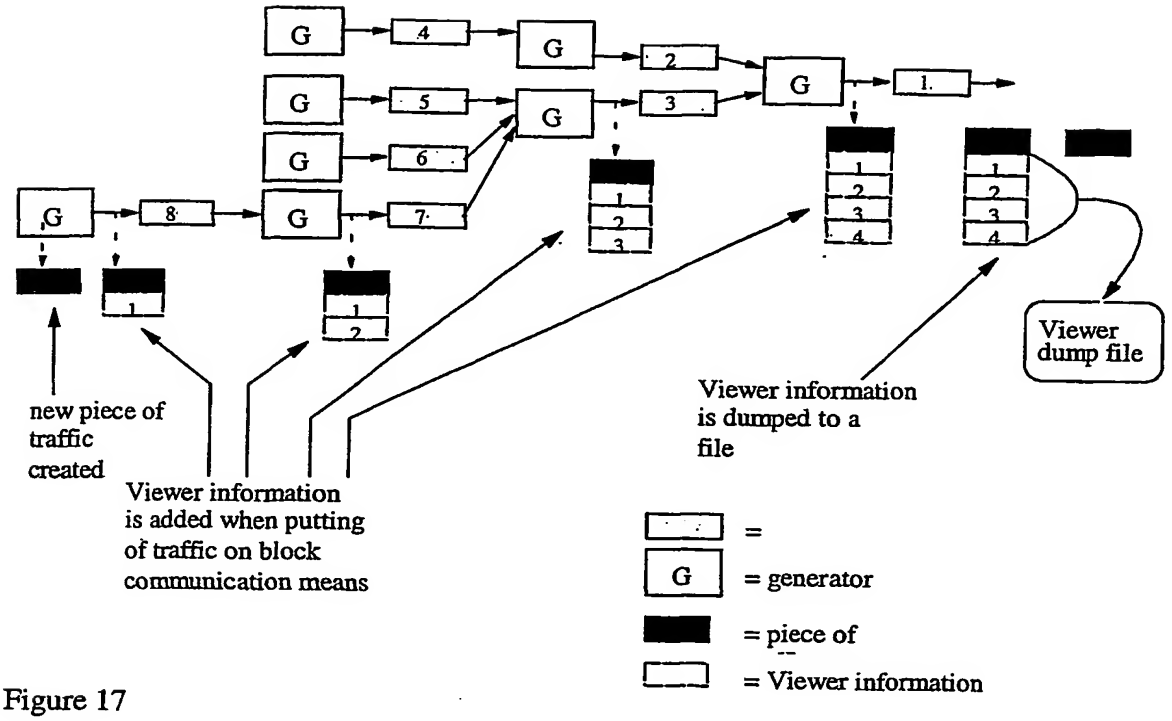


Figure 17

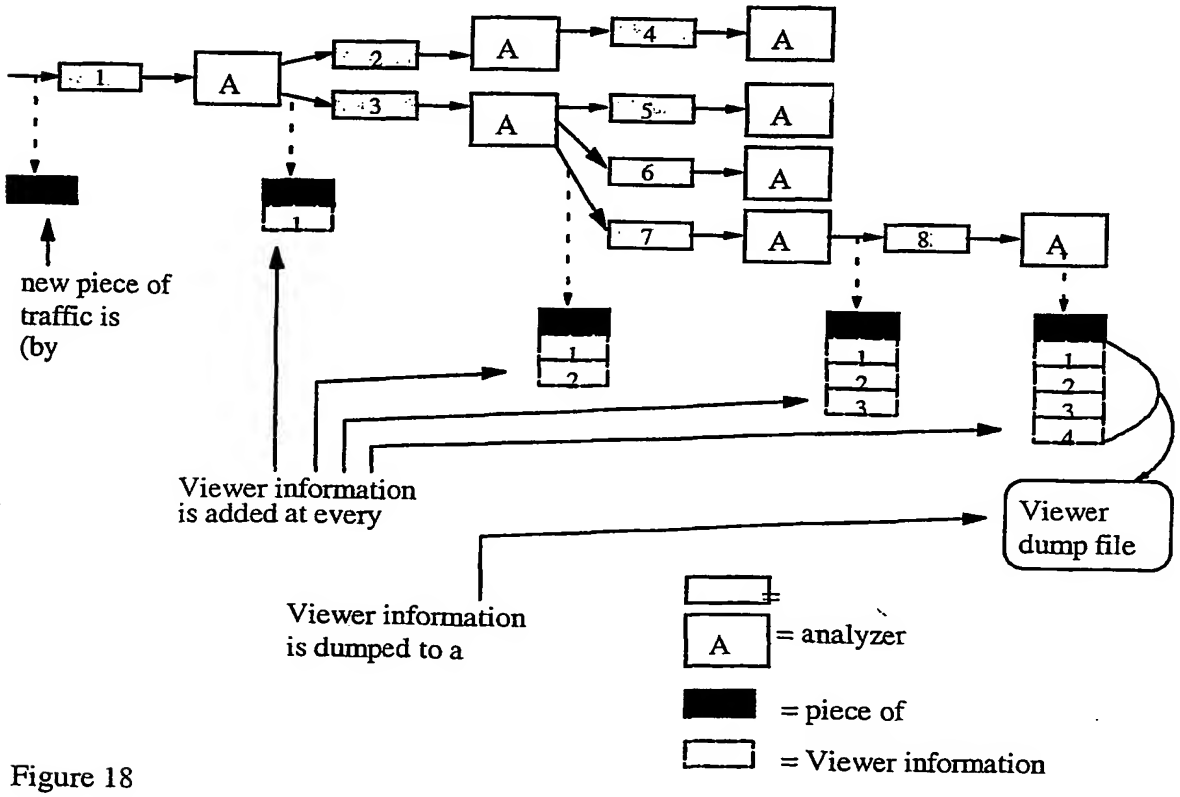


Figure 18

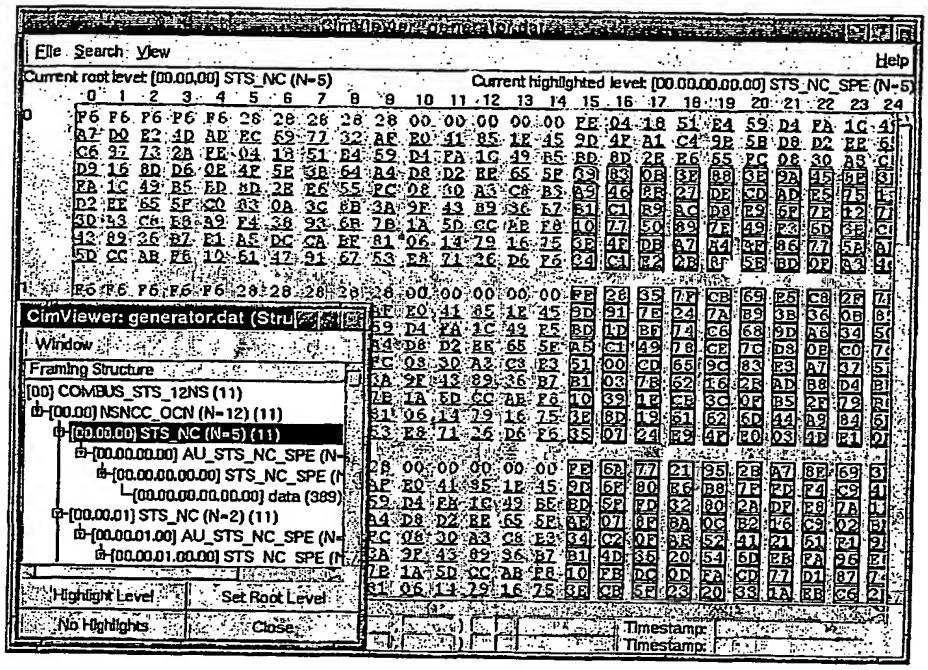


Figure 19

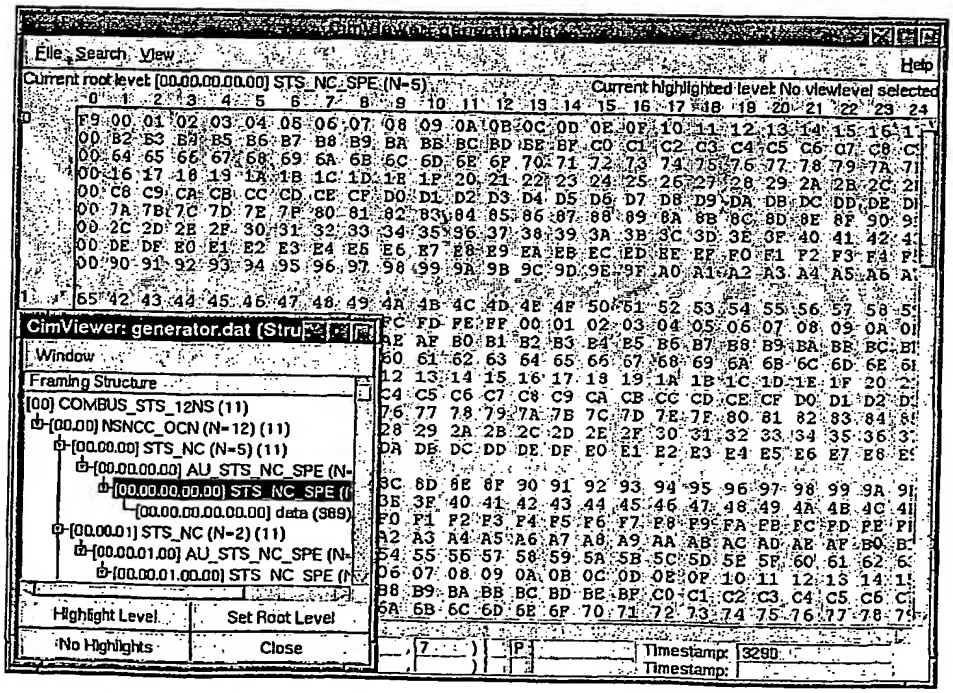


Figure. 20a

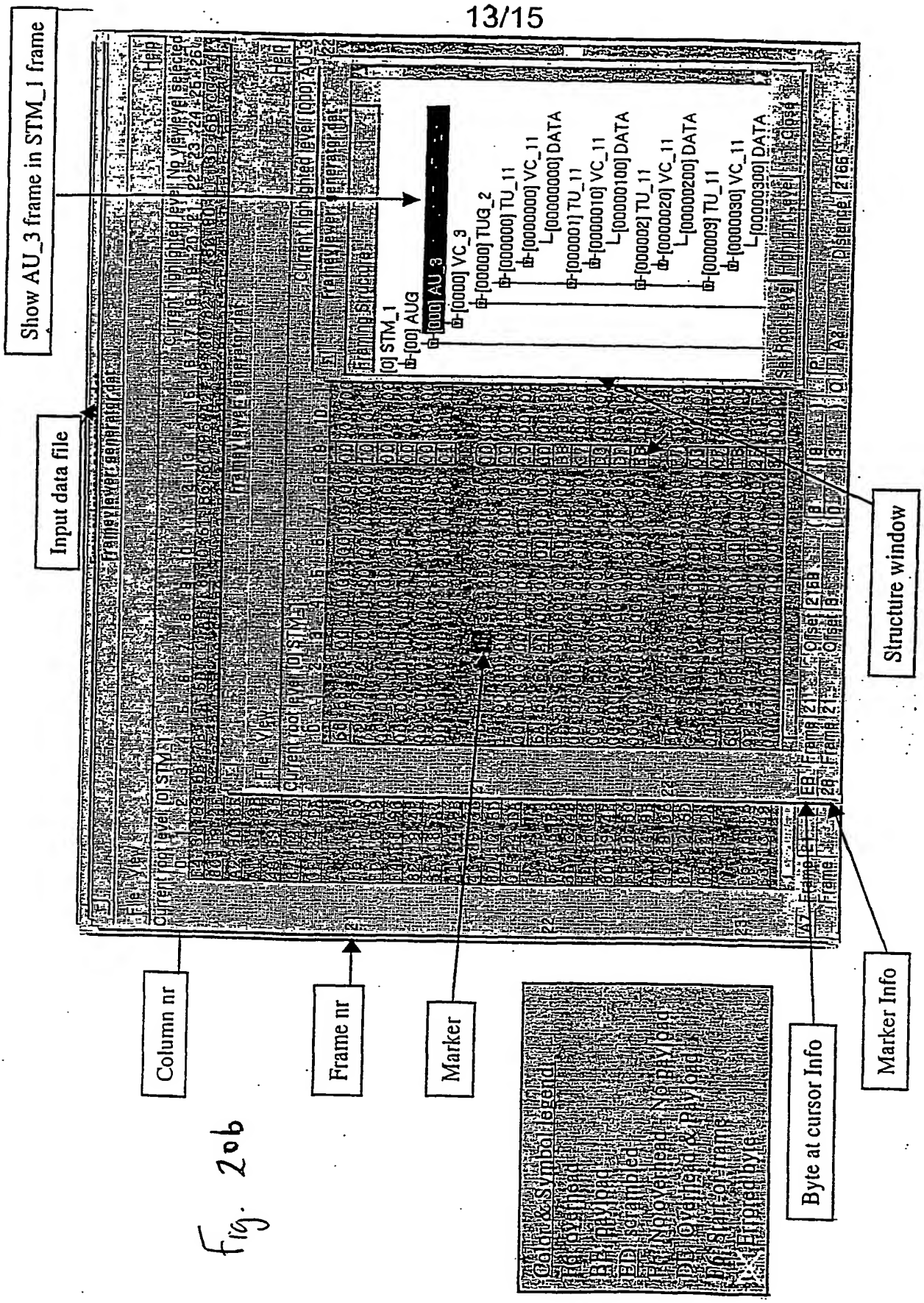


Fig. 2b

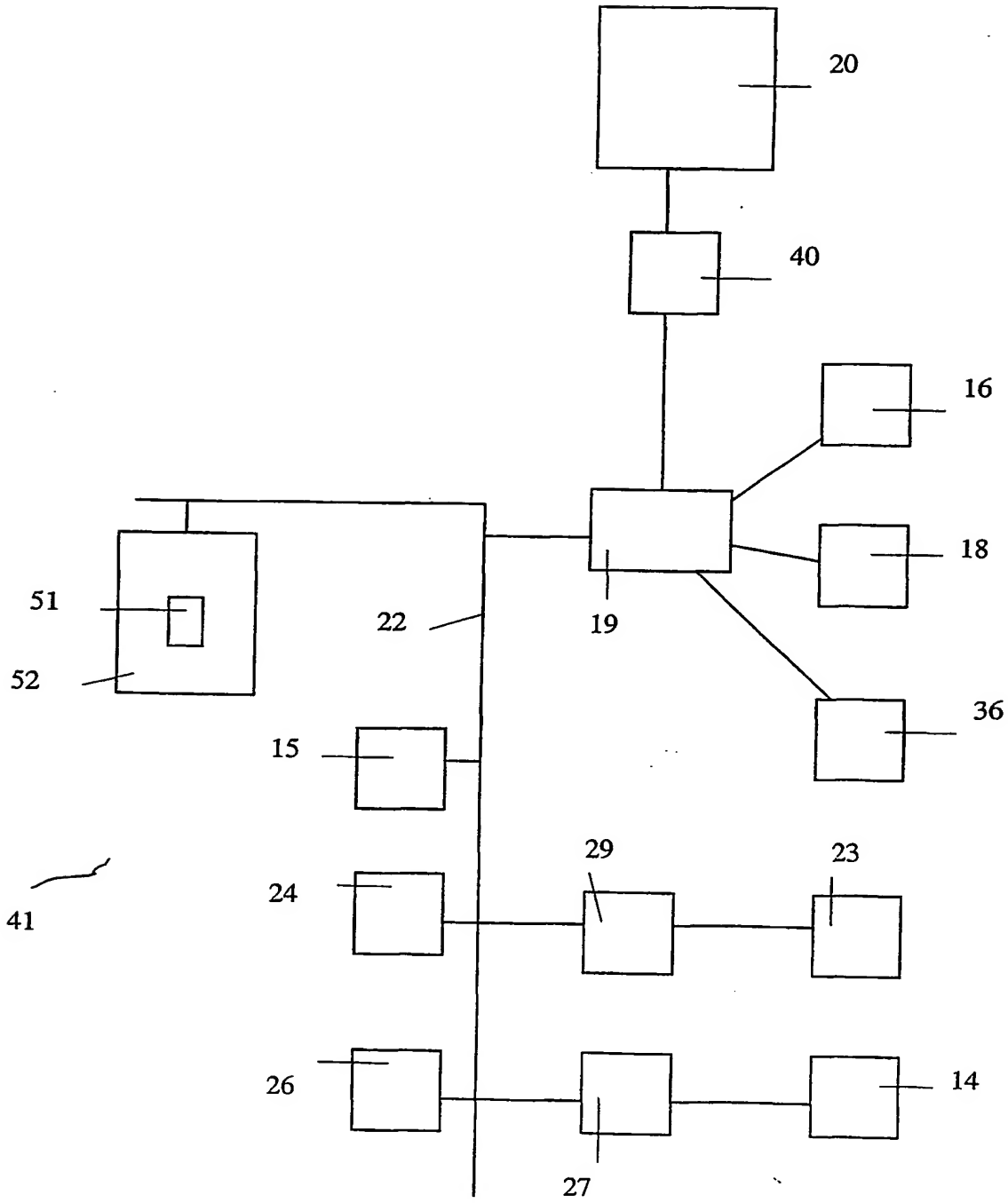


Fig. 21

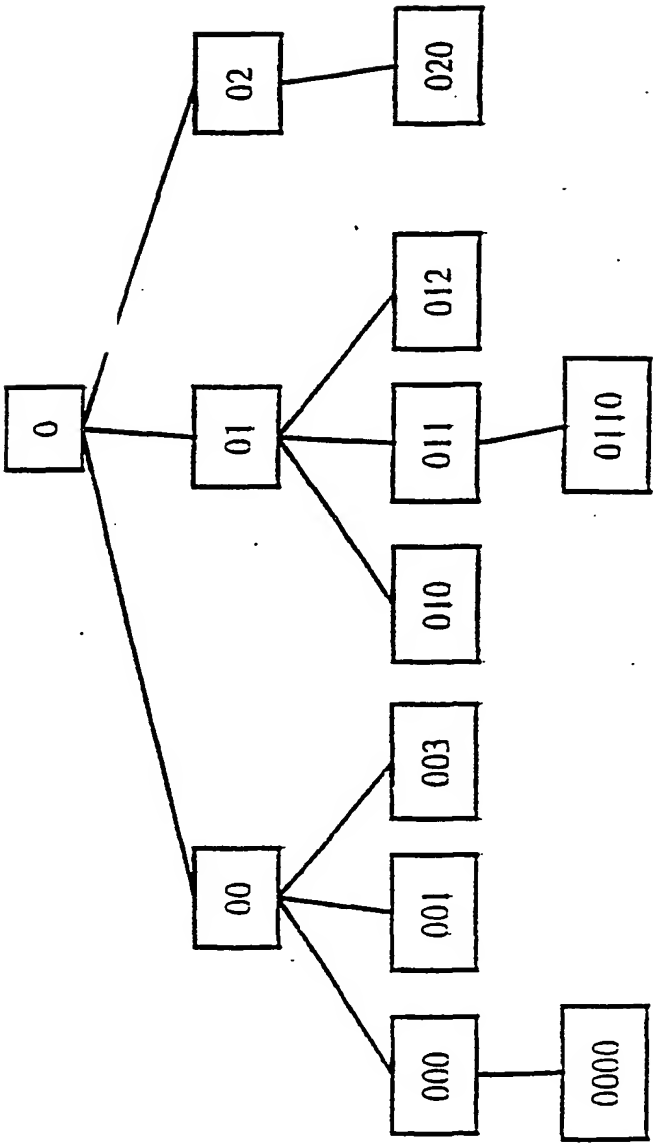


Fig. 22

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
16 May 2002 (16.05.2002)

PCT

(10) International Publication Number
WO 02/039325 A3

(51) International Patent Classification[7]: H04L 12/26, 29/06

(21) International Application Number: PCT/BE01/00193

(22) International Filing Date:
8 November 2001 (08.11.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
0027258.3 8 November 2000 (08.11.2000) GB

(71) Applicant *(for all designated States except US)*: EASICS NV [BE/BE]; Interleuvenlaan 86, B-3000 Leuven (BE).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: VANDEWEERD, Ivo [BE/BE]; Vuurkruisenlaan 1, B-3500 Hasselt (BE). COENEN, Steven [BE/BE]; Grootloonstraat 101, B-3840 Borgloon (BE).

(74) Agents: BIRD, William, E. et al.; Bird Goën & Co., Vilvoordsebaan 92, B-3020 Winksele (BE).

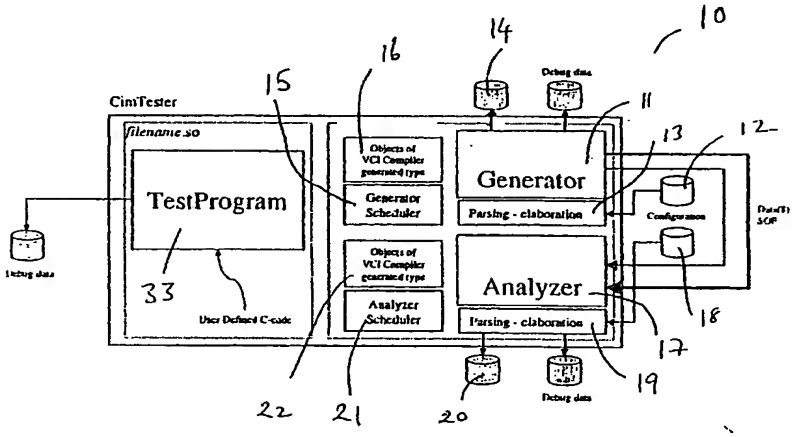
(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Declaration under Rule 4.17:
— *of inventorship (Rule 4.17(iv)) for US only*

[Continued on next page]

(54) Title: COMPUTER BASED VERIFICATION SYSTEM FOR TELECOMMUNICATION DEVICES AND METHOD OF OPERATING THE SAME



(57) Abstract: A computer apparatus is described for displaying and manipulating sequences of frames of hierarchically organized framed data. The apparatus comprises means for generating a graphical user interface on a display screen, the graphical user interface having means for displaying a representation of a hierarchy of the framed data. The computer apparatus also has a pointing device for selecting a portion of the representation of the hierarchy to generate a control signal, and means responsive to the control signal to display a portion of the framed data corresponding to the selected portion of the representation of the hierarchy. The graphical user interface may be used in a computer based verification system for the analysis and display of a sequence of frames of framed data, comprising: means for generating a frame generator for receiving the frame sequence, the frame generator comprising means for processing the frame sequence in accordance with a protocol to form a modified frame sequence and for associating with each frame a description data structure for that frame, and means for outputting the modified frame sequence and the frame description data structure.

WO 02/039325 A3

Published:
— *with international search report*

(88) Date of publication of the international search report:
3 January 2003

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

INTERNATIONAL SEARCH REPORT

Internal Application No
PCT/BE 01/00193

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 H04L12/26 H04L29/06

International Patent Classification[7]: H04L 12/26, 29/06

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 H04L G06F G01R H04M H04J

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)
EPO-Internal, WPI Data, PAJ, INSPEC, IBM-TDB, COMPENDEX

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	KIEFER R: "SDH - NEUE SYSTEME, NEUE MESSAUFGABEN" NACHRICHTENTECHNIK ELEKTRONIK, VEB VERLAG TECHNIK. BERLIN, DE, vol. 43, no. 6, 1 November 1993 (1993-11-01), pages 282-284, XP000438839 ISSN: 0323-4657 page 283, paragraph 3 page 284; figure 3 ---	1-3
A	US 5 572 515 A (WILLIAMSON ALISTAIR ET AL) 5 November 1996 (1996-11-05) abstract column 7, line 30 - line 61; figure 11 --- -/--	1-3

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
9 July 2002	26. 07. 2002

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Peeters, D

Form PCT/ISA/210 (second sheet) (July 1992)

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
A	"Serialtest ComProbe, screenshot (Frame Display)" FRONTLINE TEST EQUIPMENT, 'Online! 10 February 1998 (1998-02-10), pages 1-1, XP002200616 Retrieved from the Internet: <URL:http://www.fte.com/stc03.asp> 'retrieved on 2002-05-30! the whole document	2,3
A	US 5 920 711 A (HOLTMANN ULRICH E ET AL) 6 July 1999 (1999-07-06) abstract column 1, line 34 -column 3, line 28; figure 53	12-30
A	EP 0 397 934 A (HEWLETT PACKARD LTD) 22 November 1990 (1990-11-22) abstract column 14, line 18 -column 15, line 2; figures 4,7 column 22, line 40 - line 55 column 28, line 50 -column 29, line 5	12-30
A	"Synopsys Telecom Workbench (TWB)" CADENCE CONNECTIONS PROGRAM, 'Online! XP002205164 Retrieved from the Internet: <URL:http://www.connectionsprogram.com/memberpages/synopsys/twbtoncsim.htm> 'retrieved on 2002-07-09! the whole document	12-30
A	"Telecommunications Workbench Family" SYNOPSYS TELECOMMUNICATIONS WORKBENCH FAMILY, 'Online! 2000, pages 1-4, XP002205165 Retrieved from the Internet: <URL:http://www.synopsys.com/sps/pdf/family_ds.pdf> 'retrieved on 2002-07-09! the whole document	12-30
A	"Telecommunications Workbench Family ATM" SYNOPSYS TELECOMMUNICATIONS WORKBENCH FAMILY, 'Online! 2000, pages 1-6, XP002205166 Retrieved from the Internet: <URL:http://www.synopsys.com/sps/pdf/atm_ds.pdf> 'retrieved on 2002-07-09! the whole document	12-30

-/--

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	"Telecommunications Workbench Family SDH" SYNOPSYS TELECOMMUNICATIONS WORKBENCH FAMILY, 'Online! 2000, pages 1-6, XP002205167 Retrieved from the Internet: <URL:http://www.synopsys.com/sps/pdf/sdh_ds.pdf> 'retrieved on 2002-07-09! the whole document ---	12-30
A	"Synopsys and Nitec" SYNOPSYS SUCCESS STORIES, 'Online! 1998, pages 1-4, XP002205168 Retrieved from the Internet: <URL:http://www.synopsys.com/products/success/nortel_ire_ss.pdf> 'retrieved on 2002-07-09! the whole document ---	12-30
A	"Synopsys and Lucent Technologies" SYNOPSYS SUCCESS STORIES, 'Online! 1998, pages 1-4, XP002205169 Retrieved from the Internet: <URL:http://www.synopsys.com/products/success/lucent_ss.pdf> 'retrieved on 2002-07-09! the whole document -----	12-30

INTERNATIONAL SEARCH REPORT

International application No.
PCT/BE 01/00193

Box I Observations where certain claims were found unsearchable (Continuation of item 1 of first sheet)

This International Search Report has not been established in respect of certain claims under Article 17(2)(a) for the following reasons:

1. ☐ Claims Nos.:
because they relate to subject matter not required to be searched by this Authority, namely:

2. ☐ Claims Nos.:
because they relate to parts of the International Application that do not comply with the prescribed requirements to such an extent that no meaningful International Search can be carried out, specifically:

3. ☐ Claims Nos.:
because they are dependent claims and are not drafted in accordance with the second and third sentences of Rule 6.4(a).

Box II Observations where unity of invention is lacking (Continuation of item 2 of first sheet)

This International Searching Authority found multiple inventions in this international application, as follows:

see additional sheet

1. ☒ As all required additional search fees were timely paid by the applicant, this International Search Report covers all searchable claims.

2. ☐ As all searchable claims could be searched without effort justifying an additional fee, this Authority did not invite payment of any additional fee.

3. ☐ As only some of the required additional search fees were timely paid by the applicant, this International Search Report covers only those claims for which fees were paid, specifically claims Nos.:

4. ☐ No required additional search fees were timely paid by the applicant. Consequently, this International Search Report is restricted to the invention first mentioned in the claims; it is covered by claims Nos.:

Remark on Pr test

☐ The additional search fees were accompanied by the applicant's protest.

☒ No protest accompanied the payment of additional search fees.

FURTHER INFORMATION CONTINUED FROM PCT/ISA/ 210

This International Searching Authority found multiple (groups of) inventions in this international application, as follows:

1. Claims: 1-11, 29-30 as dependent on claims 1-11

   Computer apparatus for displaying and manipulating hierarchically organized framed data.

2. Claims: 12-28, 29-30 as dependent on claims 12-15

   Apparatus, method and computer program product for generating and analyzing frame sequences, and system for generating frame generators and frame analyzers.

INTERNATIONAL SEARCH REPORT

Internal Application No
PCT/BE 01/00193

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 5572515 A	05-11-1996	NONE	
US 5920711 A	06-07-1999	NONE	
EP 0397934 A	22-11-1990	EP 0397934 A1	22-11-1990
		WO 9014723 A1	29-11-1990
		WO 9014724 A1	29-11-1990
		JP 3004648 A	10-01-1991